AFIT/GCS/ENG/92D-24

FORMALIZING, VALIDATING, AND VERIFYING REAL-TIME SYSTEM REQUIREMENTS WITH REACTO AND VHDL

THESIS

Frank Charles Duane Young
Captain, USAF

AFIT/GCS/ENG/92D-24

AFIT/GCS/ENG/92D-24

FORMALIZING, VALIDATING, AND VERIFYING REAL-TIME SYSTEM REQUIREMENTS WITH REACTO AND VHDL

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science in Computer Engineering

Frank Charles Duane Young, B.S.C.S.

Captain, USAF

December 15, 1992

Preface

I dedicate this thesis to my wife Sharon and our children Lonnie, Andrew, Jonathan, Rachel, and Michael. Without their loving support, I would have given up long ago. I appreciate the wise counsel and guidance of my advisor, Major Kim Kanzaki, and my committee members, Major Paul Bailor and Dr. Thomas Hartrum. I also wish to thank Li-Mei Gilham of Kestrel Institute. Her professional and timely responses to my inquiries about Reacto were the key to my success with this new and promising tool. I especially want to thank Lieutenant Colonel Bill Hobart, my academic advisor, whose Barnabas-like encouragement and prayerful support kept me on track when all seemed insurmountable. Finally, thanks be to God, with whom all things are possible— even AFIT theses.

Frank Charles Duane Young

Table of Contents

List of Figures

List of Tables

Abstract

We develop a methodology for formalizing, verifying, and validating the requirements specification of real-time systems based on a graphical and formal hierarchical Finite State Machine (FSM) language *Reacto*. We define a means to quantify time and express real-time constraints in Reacto and a transformation from Reacto to the Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL). Reacto's high level abstractions, graphical nature, and theorem prover produce efficient, accurate, and easily understood specifications. We use VHDL's event driven simulation capability, concurrency, and temporal operators to thoroughly examine temporal dependencies between the state machine transitions, and to increase simulation power by simulating multiple communicating FSMs. We apply the methodology to two example problems, a cruise control, and a lift (elevator) controller. We verify that the state machine specification is consistent and validate the specification using executable simulations in both Reacto and VHDL. We evaluate the methodology against criteria for real-time specification languages and conclude that Reacto and VHDL complement each other well. Together, they abstract the real world well, are clearly understood, verify that the specification and implementation are consistent, are easy to modify, allow requirements tracing, and finally, support specification of concurrency and timing constraints.

FORMALIZING, VALIDATING, AND VERIFYING REAL-TIME SYSTEM REQUIREMENTS WITH REACTO AND VHDL

I. Introduction

1.1 Background

Traditionally, we describe the software development process using the waterfall model and diagrams similar to Figure 1. The boxes represent a series of activities connected together by products (the arrows). The product of each activity is used by the next activity. Figure 1 shows the iterative nature of this process, whereby each stage feeds back

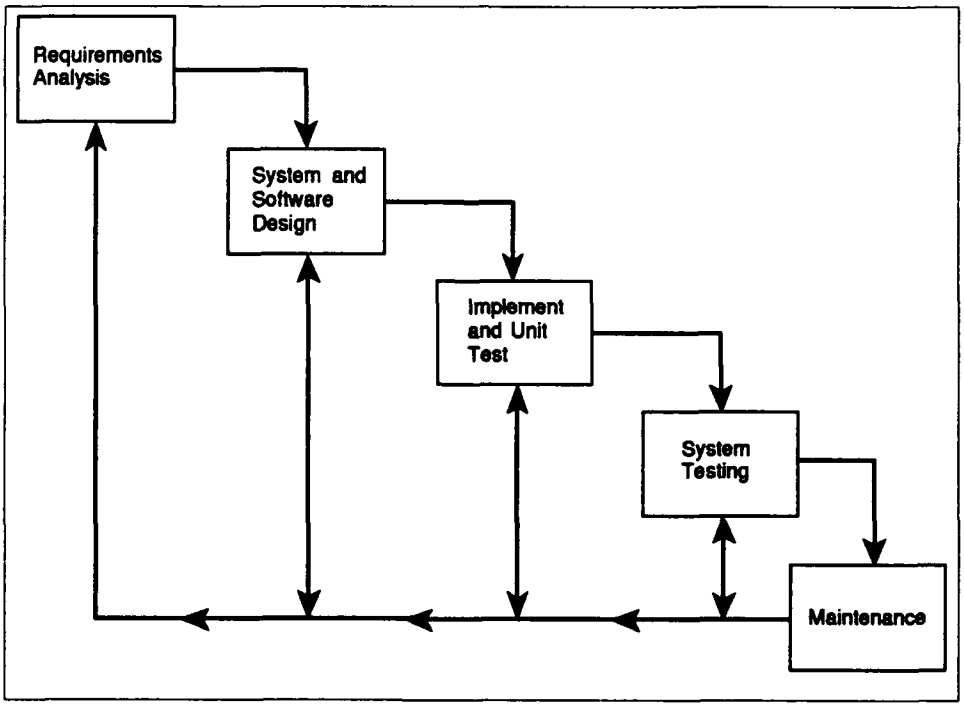

Figure 1. Iterative Waterfall Life Cycle Model (35)

into the process further refining the products in each case. At each stage, the success of an activity cannot exceed the quality of its input. Although it is theoretically possible

to produce a perfect product by iteratively refining the products many times, it is very expensive. Therefore, it is important to produce a quality product from the each activity. Generally, it is so much more expensive to fix errors discovered later in the process that we can afford to expend a great deal of effort in the earlier stages and save time and money in the long run. In fact, independent studies by GTE, TRW and IBM show that it costs up to 200 times as much to fix errors in the maintenance phase of a system lifecycle as it does during requirements analysis (9:24). In our research, we are focusing on the first activity, "Requirements Analysis", in an attempt to improve the Requirements Analysis process for real-time computer systems. A *real-time* or *reactive* system is a system which must continuously react to external and internal stimuli (14:231). Examples of real-time systems include automobiles and telephone switches, and example stimuli for these two systems include control button and electronic sensor inputs.

Requirements analysis is examining a problem to understand what the user wants or needs a system to do. It involves talking to people, discovering who they are, what they want, and the constraints they want to impose on the system (9:41). For real-time systems, these constraints include timing constraints on system behavior.

The product of the requirements analysis activity is a system requirements specification (SRS). Generally, software engineers agree that an SRS is a complete description of **what** a system will do without specifying **how** to accomplish it (9:17). The difference between what and how is not easy to define, and it depends on the level of detail with which we examine the problem. We refer the interested reader to Alan Davis' book for a detailed discussion of what versus how in system specifications (9:15-19). For our purposes, we attempt to specify what the external behavior of the system must be without constraining the developers to any particular implementation of that system (how). This is Davis' level 3 in the "what versus how" dilemma (9:18).

Ideally, an SRS correctly and completely describes the user's requirements for the final product. Generally, an erroneous SRS describes behavior that is not correct, or not in accordance with the user's desires. There are several categories of errors we might associate with an SRS. We separate these errors into four categories, incorrect facts, omissions,

inconsistencies, and ambiguities. SRS's without these errors exhibit the following qualities according to Davis.

A *correct* (contains no incorrect facts) SRS contains only requirements that represent something required of the system to be built. We cannot define this category in general, since it depends totally on the application at hand. For example, if the system must respond to all button presses within 5 seconds and the SRS states that "the software shall respond to all button presses within 10 seconds," the SRS is incorrect (9:184-185).

A *complete* (nothing is omitted) SRS possesses the following four qualities (9:188-190). First, everything that the software is supposed to do is in the SRS. It is most difficult to define or detect violations of SRS completeness. Violations are difficult to detect because they imply that something is not in the SRS. How can specifiers find something that is not present by examining what is present? Only those who own the problem to be solved by the software can detect such an oversight or omission. Second, definitions of every software response to all realizable input classes in all realizable classes of situations are included. It is particularly important to specify the responses to both valid and invalid inputs. This implies that for every system input included in the SRS, the SRS specifies what the appropriate output is. Additionally, the appropriate output may not be just a function of the input, it may also be a function of the current state of the system and the temporal relationship of that input to other inputs and outputs. Third, it is a finished document- e.g., all pages are numbered, all figures and tables are filled in. Fourth, no sections are marked "To Be Determined (TBD) (9:190)."

In a *consistent* SRS, no subset of individual requirements stated therein conflict. These conflicts show up in a number of ways (9:191):

Terms Conflict Two terms are used in different contexts to mean the same thing. For example, using the terms "prompt" and "cue", to denote a message displayed by the software to ask the user to enter some information, in different SRS sections violates consistency.

Conflicting Characteristics Two parts of the SRS demand the product to exhibit contradictory traits. For example, in one place, the SRS requires all inputs via menu, and in another place, it specifies command language inputs.

Temporal Inconsistency Two parts of the SRS demand the product to obey contradictory temporal behavior-e.g., one sentence in the SRS states that "System input A will occur only while system input B is occurring." And another place in the SRS states, "System input B may start 15 seconds after system input A finishes."

"An SRS is nonambiguous if and only if every requirement stated therein has only one interpretation. Imagine that a sentence is extracted from an SRS, given to ten people who are asked for their interpretation. If there is more than one such interpretation, then that sentence is probably ambiguous... In particular, using natural language invites ambiguity because natural language is inherently ambiguous.... (9:185)"

Incorrect facts, omissions, inconsistencies, and ambiguities that are not detected in the initial specification are sometimes detected when the system is being developed, but they are inevitably found after the system has been delivered. As mentioned before, the longer errors go undetected the more expensive they are to fix. If the cost to fix one or more errors is too great, the system may be thrown away or simply never used. In today's Air Force, we cannot afford to throw important systems away or to spend money fixing systems that don't work. We must insure that the complex real-time systems we need to maintain our technological superiority are built efficiently and accurately. To do so, we must be able to create accurate and practical SRSs for these real-time systems.

SRSs for real-time systems are different and usually more complex than those of traditional SRS (e.g., an SRS for a database or accounting system). Harel states that real-time system behavior is much more difficult to describe (14:232). A real-time system specification must not only describe the relationship between inputs and outputs, but it must also describe the relationship between inputs and outputs with respect to time. For large problems, it has been especially difficult to write an SRS that is clear and understood by humans and formal enough for computer analysis at the same time.

After we develop informal and formal SRSs we review them to determine if they are accurate, complete, consistent and unambiguous. We use the terms validation and verification to describe the processes we go through attempting to insure that a system accomplishes its requirements.

Simply, *validation* is determining if we are building (or have built) the correct product (29:499). In general, validation is checking the product of each software development process activity (each box in Figure 1) against the results of the previous activity to insure the products are consistent. Since the SRS is the product of the first activity of the software development process, validation of the SRS is checking the SRS to determine if it accurately and completely describes what the user wants the system to do (the informal specification). If each product is consistent with the product from the previous stage, then the delivered system is be consistent with the SRS. Figure 2 depicts the Requirements Analysis phase of the Waterfall model depicted earlier in Figure 1. Validation is also an

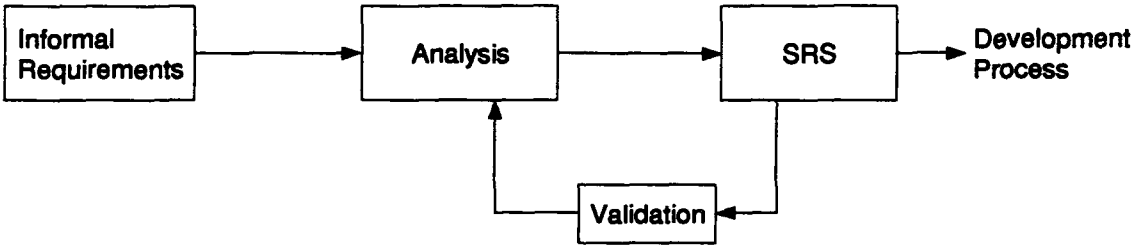

Figure 2. Requirements Analysis Process

iterative process, repeatedly clarifying and analyzing the SRS against the informal requirements until both the user and the specifier are satisfied that the SRS accurately reflects the real requirements. Ideally, there are no remaining incorrect facts, omissions, inconsistencies, or ambiguities; realistically, informal requirements are a moving target, and the best we can hope for is an accurate snapshot of the requirements. However, the better the SRS, the cheaper the system, and the better the system meets the users requirements. Therefore SRS validation is a good investment.

Simply, *verification* is determining if we are building (or have built) the product correctly (29:499). Historically, the process of checking whether or not a software product entering the maintenance activity meets the requirements is known as verification. Usually,

this involves testing the final software product against the SRS. In fact, we should write SRS's specifically to support verification. Total verification requires an error free SRS, i.e., a correct, complete, consistent, and nonambiguous SRS. Additionally, it requires an SRS free of requirements equivalent to Turing's halting problem because it would take an infinite amount of time to verify such requirements (9:191). Practically, we limit our verification efforts according to what we perceive the important requirements are, and we limit testing of difficult or intractable requirements to the time scheduled for testing.

We extend the traditional concept of verification to include the process of making assertions about the system itself in the SRS and attempting to verify the fact that the SRS does not violate those assertions. A heating system provides a convenient example. "Oil valve is closed when the pilot light is out" is an assertion which the the heater controller must fulfill. If we detect a contradiction in the heater controller SRS such that the oil valve might be open under some condition while the pilot light is out, the assertion would be false, and the SRS unverified. If however, we can prove that the oil valve is closed at all times when the pilot light is out, then the assertion is verified.

1.2 Problem Statement

We will investigate the feasibility, benefits of and the problems associated with formalizing, validating and verifying real-time SRSs using Reacto and the Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL). Additionally, this requires defining a mapping and translation from Reacto to VHDL.

1.3 Research Objectives

This research is part of a larger effort to formalize and automate the entire software development process. Eventually, we plan to be able to formally specify systems and guide the automated implementation of that specification. We contend that the specified behavior can be achieved in a much more efficient, reliable and reproducible way (4:3). Such a capability can make the specification itself the target of maintenance as shown in Figure 3.

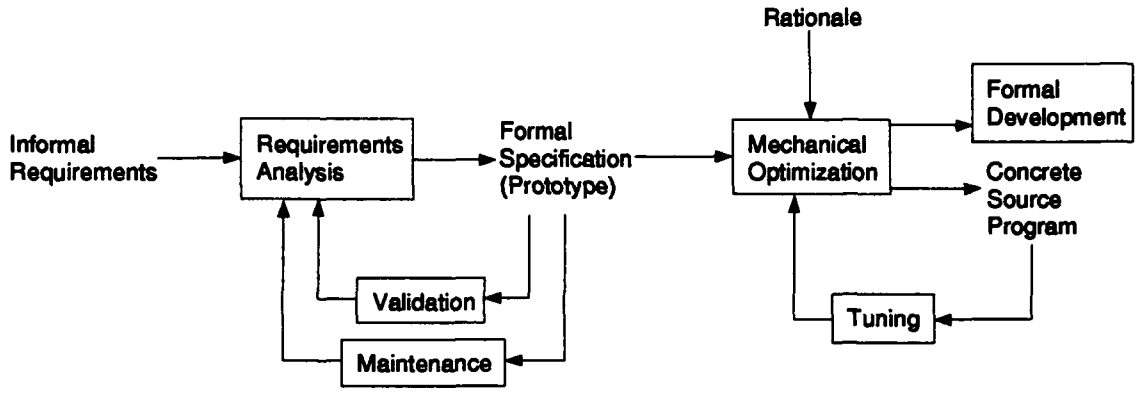

Figure 3. Formal Software Development Process

Our Program Transformation System Paradigm can significantly reduce the need to understand and maintain source code since it can be automatically regenerated from the updated specification (4:4). "Tuning" as shown in Figure 3 might be necessary to enhance efficiency, but is not necessary to correct behavior.

One of the requirements of such a paradigm is a complete and accurate specification. Hence, we need a means to formally specify and examine system behavior. Real-time systems are among the most complex systems, and at the same time, one of the most common types of systems used in the Air Force. Hence, the target of this research effort is to improve our capability to specify, validate and verify real-time system behavior. Such a capability could be of considerable use in the development and maintenance stages of real-time systems.

We plan to build a formal model of the SRS in both the Reacto and VHDL languages and verify and validate the model against the informal requirements specifications. Previous efforts to formalize and validate SRSs here at The Air Force Institute of Technology (AFIT) focused on using the wide spectrum[1] language Refine[2] and VHDL in parallel efforts on identical problem descriptions. We observed that the two languages complement each other, Refine being more abstract and well suited to ironing out behavioral problems quickly, and VHDL for looking at behavioral details, especially temporal details. We hope

[1] A wide spectrum language allows descriptions of behavior at both high and low levels of abstraction.
[2] Refine^{TM} is a trademark of Reasoning Systems, Inc.

to capitalize on our observation by using Reacto (based on Refine) to quickly create a formal SRS, translate that SRS into VHDL, and examine the SRS in detail in VHDL.

In addition to the results of our previous research, we have several specific reasons for choosing Reacto and VHDL in this effort. Reacto provides a means to formally organize system behavior (i.e., the SRS) using a hierarchical Finite State Machine (FSM). Reacto's visual interface makes it is easy to specify and test behavior. Reacto allows a high level of abstraction, and because it uses the intuitive notion of a state machine in its formalism, it can be readily understood by users. Reacto includes an automated verification system for formal verification of Reacto state machines which we use to verify the SRS as described in section 1.1. Reacto is based on the Refine language which provides the capability to generate an automated transformation from Reacto to VHDL. VHDL is based on time, and it provides a large set of operations to manipulate, investigate, and understand time and the effects it has on behavior. VHDL is flexible enough to specify behavior in a multitude of ways, including a state machine formalism. VHDL is a concurrent language allowing us to simulate the behavior of concurrent components- e.g., multiple state machines.

Our specific objectives include:

1. Define a means to map time constraints into Reacto.
2. Define a means to validate time constraints in Reacto.
3. Specify a transformation from Reacto to VHDL.
4. Specify, verify, simulate, and validate the behavior of a both a cruise control and a Lift system in Reacto and VHDL.
5. Compare Reacto and VHDL capabilities.
6. Recommend enhancements to Reacto and VHDL.

1.4 Scope of Effort

We are not attempting to verify or validate the systems completely. While that is our future goal, this research is simply a step in that direction.

We are attempting to demonstrate a "proof of concept." If we demonstrate significant benefits of the methodology, we can pursue actually generating an automated transformation and or enhancing Reacto with some of VHDL's capabilities.

We are specifying a partial mapping from Reacto to VHDL, but not a general mapping for the complete Reacto language.

1.5 Equipment and Software

We use Reacto version 2.0, running on the software engineering SPARCstation[3] 2 Network, for the Reacto portion of this research. And, we use Synopsis[4] version 2.2a for the VHDL portion.

1.6 Order of Presentation

First, in Chapter II, we discuss some of the properties of specification languages desired for real-time systems. Specifically, we discuss specifying timing constraints and discuss the benefits and detriments of specifying concurrency. Next, we consider some example requirements specification languages that are used to specify real-time systems. We then discuss in some detail previous efforts to use Refine and VHDL to specify real-time system behavior here at the Air Force Institute of Technology (AFIT). Finally in the literature Review, we introduce Reacto an important part of our research. We conclude by summarizing some of the features of specification languages for real-time systems we intend to provide in our methodology.

In Chapter III, we explain in detail the method we apply to improve requirements specification of real-time systems using Structured Analysis and Design Technique (SADT[5]), Reacto, and VHDL. Additionally, we introduce two sample problems which we use to demonstrate the methodology.

In Chapter IV, we discuss the application of Reacto to the two sample problems. We introduce the time extensions we add to Reacto to model timing requirements. We

[3]SPARCstation^{TM} is a registered trademark of SPARC International, Inc.
[4]Synopsis^{TM} is a registered trademark of Synopsis, Inc.
[5]SADT^{TM} is a trademark of SofTech, Inc.

describe the test cases used to verify and validate the Reacto specification, and we discuss refining the specification using Reacto, enumerating many specification improvements.

In Chapter V, we discuss the Reacto to VHDL transformation, describing a mapping between the languages and showing example transformations. This process lends insight into the differences between the Reacto and VHDL languages, and the difficulties of describing the relationships of system inputs and outputs with respect to time.

In Chapter VI, we discuss the application of VHDL to the two sample problems. We discuss how VHDL simulations work and increase simulation power. We describe special VHDL test cases and further specification refinements as a result of the VHDL simulation.

In Chapter VII, we compare the Reacto and VHDL simulation results and language similarities and differences. We summarize the benefits and limitations of the two languages.

In Chapter VIII, we summarize the results of our research and make some recommendations for future research using Reacto and VHDL. Finally, we discuss some lessons learned.

II. Literature Review

In this chapter, we provide background information about specification languages for real-time systems. We examine some properties desired of all specification languages, and then focus on the problems of specifying concurrency and timing constraints. Then we examine two commercial languages that have been used to specify real-time systems, looking for their strengths and weaknesses. The first language is not based on state machines, the second is. After examining these two languages, we review two research projects here at AFIT involving new approaches to formally specify and validate real-time system requirements. The last section introduces the Reacto state-based specification language, describing its main features, preparing the reader to understand our extension and application of it in this research.

2.1 Requirements Specification Languages For Real-Time Systems

For many years, software engineers have been searching for ways to discover and eliminate specification errors and ambiguities for all types of systems. They have tried to develop graphical and textual specification languages that are convenient to use, and well understood by users, engineers, and developers and at the same time formal enough to support computer analysis. There are several requirements specification languages available today that work fairly well for given classes of problems (38). For the most part, these languages are informal; they depend to some degree on human interpretation and analysis to discover errors and ambiguities. The more formal the language, the more amenable it is to interpretation and analysis by computer. Some of the features the experts say all languages should have are (5, 38):

- Abstract real world well.
- Clearly understood by the specifier, implementer and user.
- Support verification that the specification and implementation are equivalent.
- Easy to modify and manipulate.
- Allow tracing of requirements.

Additionally, real-time systems have some special characteristics that require additional features (16, 28, 37, 41, 42, 43):

- Executable specifications.
- Support specification of concurrency.
- Support specification of timing constraints.

Zave says an executable specification language is a specialized programming language that should provide a foundation for an efficient software development process (41:212). It should satisfy three requirements. One, it must specify the functional (behavioral) requirements of the system. Two, it should support formal reasoning that validates consistency with itself, performance requirements, test cases, and global invariants. Three, it must lead to an implementation that meets the systems performance and resource requirement, recording the implementation details during the development process (41:212). It is a working model of the requirements, and it should not model the system so that it forces the designer to a particular implementation. For example, defining the structure of a system is a step towards implementation, and in general it should be avoided (9:239)(11:798).

Since there is some controversy about specifying concurrency and problems associated with specifying time constraints, we explore those two subjects in Sections 2.1.1 and 2.1.2.

2.1.1 Specification of Concurrency Is it smart to specify concurrency? How much is concurrency a function of implementation, or is it an inherent part of some applications?

Some authors insist that it is rarely necessary to specify concurrency. For example Ibrahim, Ogden, and Williams assert that specifying concurrency in real-time system requirements documents is a mistake, robbing designers and implementors of the flexibility to provide a solution that makes efficient use of underlying computer resources (18:247).

Other authors, like Harel maintain that real-time systems are naturally concurrent and distributed, requiring the ability to specify concurrency (16:404). Drusinsky and Harel maintain that a natural ability to describe concurrency is essential, and it keeps sequential descriptions from being an awkward exception (11:800).

Ibrahim, Ogden and Williams say that if there is no choice except concurrency, then concurrency should be specified (18:269). To avoid specifying concurrency when only performance is required, software engineers should carefully consider the difference between *what* needs to be done (the specification) and *how* something should be done (an implementation), specifying only essential temporal ordering.

Many specification languages today, for example Petri nets, SYSREM, GYPSY, PAISLey, Core, SADT and LOTOS allow the user to specify if operations are order dependent, but they do not constrain implementors to concurrent implementations. Hence, they specify what must be sequential, not what must be concurrent (18:251).

2.1.2 Specification of Timing Constraints In an article he wrote to clarify some of the misconceptions about real-time computing, Stankovic says, "The fundamental challenge in the specification and verification of real-time systems is how to incorporate the time metric (37:13)." He elaborates, saying that including a time metric creates problems for concurrency models and complicates verification (37:13). He says that we need to quantitatively analyze deadlines and repetition rates rather than qualitatively analyze eventual satisfaction in our specifications and designs.

Dasarathy examines what constructs are needed in requirements specification languages to quantitatively express timing constraints and how automatic test systems can validate systems that include timing constraints (8:80). He defines two categories of timing constraints for real-time systems (8:81):

- *performance constraints* are limits on the response time of the system.
- *behavioral constraints* are rates at which the environment applies stimulus to the system.

Together, these two categories express the system's capabilities and limitations from the system and environment points of view.

Dasarathy defines three kinds of temporal restrictions (8:81):

- *Maximum* means that there can be no more than t time between events.

- *Minimum* means that there must be at least t time between events.
- *Durational* means an *event* must occur for at least t time[1].

Dasarathy defines an *event* as "... a stimulus to the system from its environment, or as an externally observable response that the system makes to its environment (8:81)." Dasarathy's example of a durational temporal restriction is a requirement to hold a button down for a specified number of seconds (8:85).

For **both maximum and minimum** temporal restrictions, Dasarathy defines four types of timing constraints, based on both the system and environment points of view (8:81-84). We illustrate these with examples from a telephone system, where the telephone user creates environmental stimulus for the telephone system.

- *stimulus-stimulus* The time between consecutive input events. For example, the time between dialing digits.
- *stimulus-response* The time between an input event and the responding system's output event. For example, the time between receiver going off hook and the dial tone.
- *response-stimulus* The time between a system output event and the next input event from the environment. For example, the time between the dial tone and the dialing of the first digit.
- *response-response* The time between consecutive system output events. For example, the time between telephone rings.

Davis simplifies Dasarathy's four types of timing constraints to two types (9:323). If both the system and its environment can be specified as cooperating processes, then we only need stimulus-response and response-response constraints, because the other two timing constraint types are just these two types from the environment's point of view.

Both Dasarathy and Davis show by example that timers can be set and checked to verify stimulus-response and response-stimulus constraints via a finite state machine.

[1] Unlike Dasarathy's definition, we use the VHDL definition of an event— a change in value, thus it has no duration.

Examples from Davis are shown in Figure 4. Figure 4a shows a one-step stimulus-response

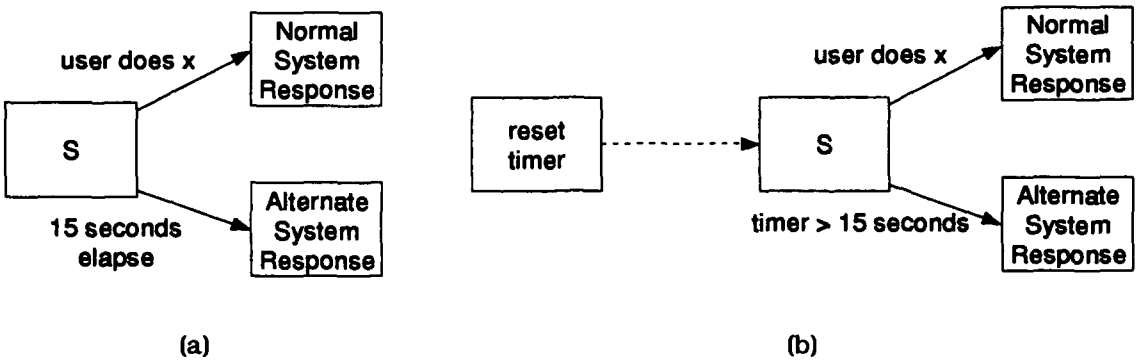

Figure 4. FSM Stimulus-Response Timing Constraint (9:324)

constraint checking mechanism, adding an extra state for the alternate (error) response. Figure 4b shows the same constraint, but there is no longer a single step between the stimulus and the response, so a timer is added to measure the stimulus-response time.

Dasarathy proposed the use of a "Latency" feature to test maximum stimulus-stimulus and response-response constraints (8:82). His Latency feature is implemented by a "wait for event" statement in the testing language. It associates system response events with a maximum wait time, triggering an error when the maximum wait time is exceeded (8:82). Similarly, he defines a "Delay" feature, to test minimum stimulus-stimulus and response-response constraints (8:85). Davis shows using a timer, and another error state "T" to model measuring the amount of time since a previous event (a response-response constraint is the time between two output events) as depicted in Figure 5. Figure 5a depicts measuring a minimum response-response time constraint, and

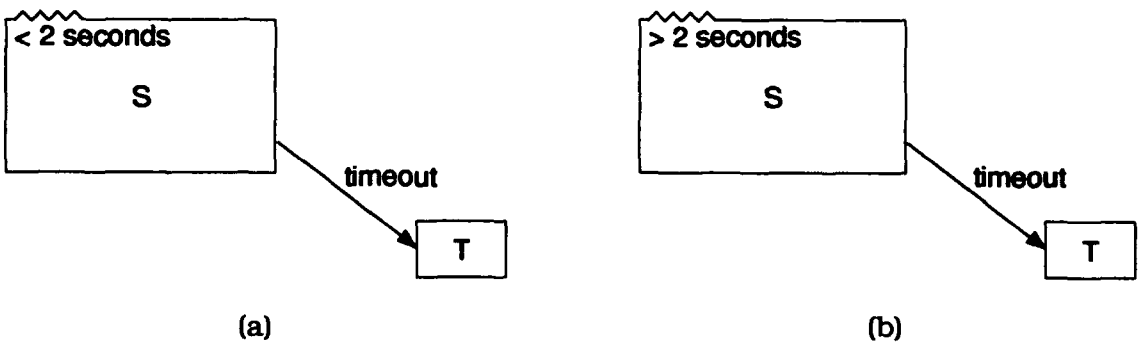

Figure 5. FSM Response-Response Timing Constraint (9:326)

Figure 5b depicts measuring a maximum response-response time constraint.

There are many other formalisms for expressing time, and specifying time constraints. Among them, Allen describes intervals and interval calculus in (2, 3). Ladkin expands Allen's intervals to unions of convex intervals in (24). And Jahanian discusses using Real Time Logic in (21, 20). These time formalisms may provide more detailed expression of and reasoning about time, but few have applied them in conjunction with a state machine formalism. In *Temporal Logic Case Study*, Wood describes applying temporal logic to a simple[2] lift problem in a FSM specification, taking a few months to understand and verify the specification (39).

2.2 Example Requirements Specification Languages

2.2.1 PAISLey PAISLey is an executable language for specifying digital systems. Zave and Schell claim it has the following desirable features:

- Supports maximal (i.e., both synchronous and asynchronous) parallelism, without mutual-exclusion problems. Abstractly, asynchronous computations are done in separate concurrently executing processes; synchronous parallelism is modeled within one process step by step keeping the specification implementation independent (43:314-315).
- Encapsulated computations. Each computation can be abstractly described as a black box with well defined connections to the outside world. This isolation, in conjunction with PAISLey's formal representation of control, enables execution of the specifications (43:316).
- Toleration of Incompleteness. Here the authors speak of *completeness* in the sense of closure; i.e., the domain and range is defined, but the mapping between the two for a given value need not be completely defined. The authors maintain that this feature enables testing of what has been said without interference from what has not been said, and that it provides a means to demonstrate the specification to customers at any stage in its development (43:317).

[2]One lift, three floors.

- Timing Constraints. Allows the user to specify any or all of an upper bound, lower bound, or distribution. Provides the means to examine performance and feasibility through PAISLey simulations. The authors admit that there is no formal means to specify traversal times through several processes (e.g., upper bounds on pipe-line delay) (43:318).
- Bounded Resource Consumption. Because of PAISLey's bound on the amount of space needed to store any process, and thereby its bound on the storage requirement for the entire system's state, specifications written in PAISLey are guaranteed to take bounded amounts of space and time (43:320).
- Consistency Checking. Well defined specifications are internally consistent. Specifications that are not well defined cannot be built (43:322).
- Coherence. Zave and Schell claim PAISLey is a simple and coherent language (43:322). Set expressions and three operators combined with mapping operators, a replication notation, and timing constraints fully define the language. And the simplicity makes it easy to learn, execute and analyze. They admit that it may be less readable than more complex languages that uses distinct syntax for familiar phrases (e.g., a CASE statement.)

While defining the differences between a programming language and a specification language, the authors state that "...a specification language must be an abstraction of all possible implementations, while a programming language has no such requirement (43:316)."

Zave and Schell assert that a specification is inconsistent if no set of scheduling decisions can avoid the violation of some timing constraint. The authors state that PAISLey demonstrates valid stimulus-response time properties of a system by ensuring the process-level timing constraints are satisfactory. They claim this can also be done analytically in some cases and by simulation in any case (43:319-320).

Commenting that validation has traditionally been done by inspection, the authors note that PAISLey goes beyond inspection to execution of the specification, thus insuring its functional and temporal validity (43:323).

Evaluating PAISLey in a more current article, Zave says that it is well suited to highly concurrent systems, but less suited to applications outside that domain (41:214). She says it is best suited for describing systems that interact with statically configured objects in its environment[3] (41:215). She says that PAISLey formally validates functional properties of real-time systems in a limited sense, lacking proof rules concerning PAISLey semantics (41:218). Zave states that PAISLey leaves open the implementation choices on how best to handle timing constraints to the developers, not constraining them to any particular implementation (41:219). And, she says that PAISLey helps construct and validate specifications, but it does not help with actually turning the specification into an implementation by automatically generating target code or some other translation process (41:220). Finally, she comments that PAISLey has been used widely in academia, but not in industry, blaming that fact on the lack of understanding most people have concerning the benefits of using an executable specification language to formalize system requirements (41:221).

2.2.2 STATEMATE Traditional Finite State Machines (FSMs) can be described as Mealy or Moore machines. Outputs of a Mealy machine are a function of the transitions, and outputs of a Moore machine are a function of the FSM's current state. Mealy and Moore machines are equivalently powerful, and one type of FSM can be converted to the other. While both Mealy and Moore FSM's are useful for specifying simple behavior in an intuitive way, they have several limitations that keep them from being used to specify large real-time system applications:

- They are "flat", having no depth, hierarchy, or modularity resulting in "state explosion" (11:804).
- They inherently impose sequential behavior (28:359).
- They cannot express concurrent behavior (3).

In his *On Visual Formalisms* article, Harel introduces hierarchical statecharts, based on Higraphs, which are derived from graphs, Euler circles, and Venn Diagrams. He proposes

[3]Meaning that PAISLey's data structures are static, making it impossible to change their sizes dynamically.

that statecharts eliminate all three traditional FSM deficiencies (15). He says, "Higraphs are suited for a wide array of applications to databases, knowledge representation, and most notably, the behavioral specification of complex concurrent systems using the higraph based language of statecharts (15:514)."

He notes that graphs are especially well suited for representing a set of related elements, and that Euler/Venn diagrams represent collections of sets and a structural (i.e., set-theoretical) relationship between them. He notes that complex computer applications managing objects, systems and situations between them can be best described using both graphs and Euler/Venn diagrams. One of the relationships inherently difficult to represent efficiently is the Cartesian product of some sets. He proposes that the Higraph representation eliminates the exponential growth in size required to represent cross-product relationships (15:515).

He claims the most beneficial application of Higraphs is to extend traditional state-transition diagrams to attain a "statechart", which can then be used to efficiently and rigorously describe the behavior of a real-time or reactive system[4]. (15:521). The problem is that the reactive behavior must be clearly described such that humans can understand it, and at the same time it must be formal and rigorous enough to allow precise computerized analysis. Harel adds a fourth deficiency to traditional FSMs saying that they have uneconomical transitions[5] (15:522).

Harel maintains that statecharts aren't flat; since they're *hi*graphs, they are hierarchical by default. Quantitatively evaluating the state explosion problem, Drusinsky states that an n-state deterministic statechart can describe a problem which takes a 2^{2^n} states in the smallest deterministic traditional FSM (11:804). Statecharts solve the state explosion problem by orthogonality. Statecharts handle concurrency by allowing output events to cause a transition in another part of the statechart (15:523). Users also claim that statecharts allow concurrency via orthogonality (34:53). Consequently, not specifying concurrency specifies sequential behavior (34:53).

[4]Harel defines a real-time system as an event driven system, continuously reacting to external and internal stimuli.

[5]The same event forces us to specify many transitions, each represented by a separate arrow from different states.

Summarizing, Harel says, "...the intricate nature of a variety of computer-related systems and situations can, and in our opinion should be represented by *visual formalisms*: visual, because they are to be generated, comprehended, and communicated by humans; and formal, because they are to be manipulated, maintained, and analyzed by computers. (15:528)."

In order to apply his statechart visual formalism, Harel has implemented the STATEMATE[6] working environment for specifying and developing complex real-time or reactive systems. STATEMATE descriptions are three separate views *structural, functional,* and *behavioral* (16:403). The structural view is a hierarchical decomposition of the system into modules and the information that flows between them including control signals (16:404). The functional model hierarchically describes system activities, data items and in addition to what is normally a functional decomposition, control activities that specify behavior (16:404). Third, the behavioral view specifies the behavior of the control activities (16:404). STATEMATE implements the graphical language statecharts to specify the behavior of control activities. This solves the traditional FSM "flatness" and concurrency problems, and makes transitions economical (16:406).

Harel claims STATEMATE descriptions are easily produced, analyzed and modified by humans (16:404). They are also formal enough for computerized validation, simulation, and analysis during any development phase (16:403).

STATEMATE provides static and dynamic analysis of the system. Static analysis includes consistency and completeness checks, comparing the three different views of the system (16:409). Dynamic analysis includes running the system through scenarios interactively, and observing the system's behavior. During dynamic analysis, users can modify the system description and rerun the scenario to check for corrected behavior (16:410). Users can also examine dynamic behavior in a non-interactive simulation using Simulation Control Language (SCL) programs. Simulations can be performed on small parts of the system or the entire system. Dynamic simulations are valuable tools for examining time-critical performance and efficiency constraints (16:411).

[6]STATEMATE^{TM} is a registered trademark of i-Logix, Inc.

In STATEMATE, actions are accomplished by executing transitions, entering or exiting states, or by being in a state. It combines the capabilities of Mealy and Moore machines together (11:798). STATEMATE also has the capability to execute scenarios involving multiple statecharts (16:410). STATEMATE can produce VHDL code as a step towards "silicon compilation" of hierarchical state machines for hardware designers (16:412).

Smith and Gerhart comment on the fact that STATEMATE is unable to model large numbers of similar activities (like a collection of identical bank-teller machines) (34:54). They note that Harel has proposed extensions to STATEMATE to handle the problem, but his extensions don't address the problem of distinctly referencing individual activities (34:54). Alagar and Ramanathan say that the statecharts formalism "...does not demonstrate formal reasoning based on the diagrammatic description (1:254)," insinuating that the verifications performed do not actually verify everything described in the statechart diagrams.

2.3 Douglass and Eickmeier

Randy Douglass and Dan Eickmeier researched transforming informal specifications into formal executable SRSs at AFIT in 1991. Douglass defines a method to transform a specification from its graphical and textual representation into Refine (10). He transformed two problem specifications (a home heating system and a lift system) from their informal written English specifications into SADT and finally into Refine. He used Refine simulations to discover and correct specification errors and ambiguities. He also examined the benefits from the translation and simulation process. Eickmeier defined, used and evaluated a methodology like Douglass' to transform and execute the same two problem specifications in VHDL (12).

Because our research is related to Douglass and Eickmeier's efforts, we describe the SADT, Refine and VHDL languages in Sections 2.3.1, 2.3.2, and 2.3.3. Then, we follow up with their conclusions in Section 2.3.4.

2.3.1 SADT Douglass and Eickmeier used SADT as an intermediate informal specification language between the written English specification and the formal VHDL and Re-

fine specifications for their research. Douglas Ross introduced SADT in the late 1970s (7). SADT has evolved since then, and Ross discusses SADT's metamorphosis in (31, 32, 33).

SADT is a graphical and textual specification tool for software systems. Ross (33) describes it as a "blueprint" for software. The SADT "blueprint" is intended to clearly communicate the necessary system requirements for successful implementation by a skilled programmer.

Functional SADT diagrams are similar to the traditional dataflow diagrams. Figure 6 is an example SADT diagram. The labels in Figure 6 illustrate the meaning of each

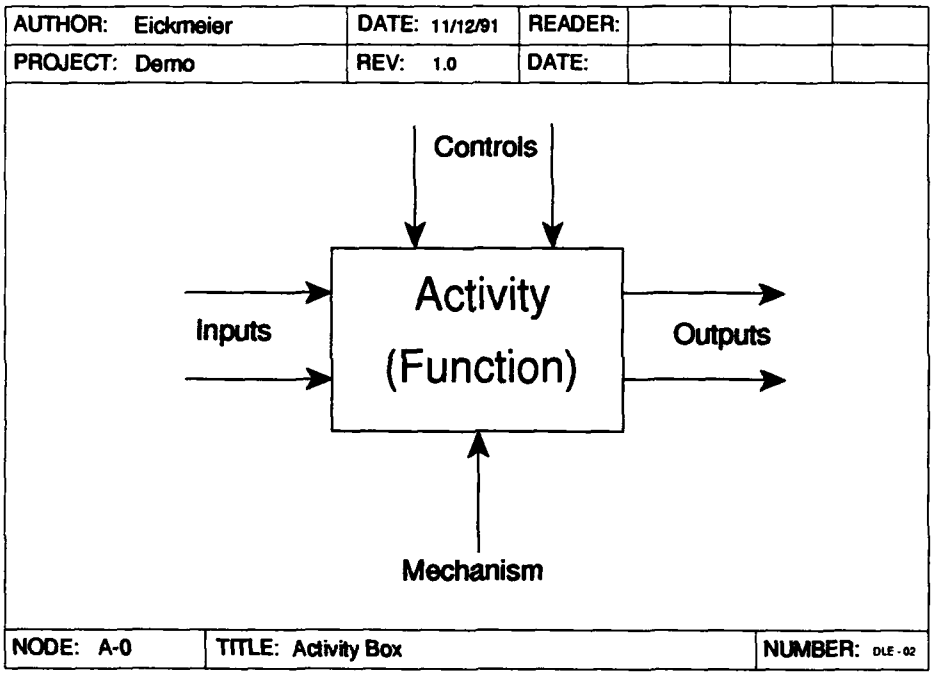

Figure 6. Sample SADT Diagram (12:11)

graphical object. Boxes in SADT diagrams represent activities or functions of the software system. For example a box labeled "Check-Password" represents the activity of verifying a password. *Input* arrows entering activity boxes from the left represent data flowing into the box. Continuing with the Check-Password example, inputs labeled "Userid" and "Password" are expected. *Controls* are inputs that constrain the activity. If our Check-Password activity allows system managers to set the length of passwords, "Password-

Length" is a *Control* input. The *Mechanism* arrow represents an already existing entity[7] (like a system library routine) that is known to be available and that is used to perform the activity. Finally, *Output* arrows represent the product of the activity, e.g., our Check-Password activity might produce a boolean flag "Access-Granted."

SADT diagrams are usually limited to six or seven activities per diagram, and like dataflow diagrams, they are hierarchical. Users describe sub-activities of complex activities in additional SADT diagrams, until at the lowest level further decomposition is not necessary.

Relative position of activities within SADT diagrams does not imply sequential behavior for the connected activities. Behavioral details for each activity are specified informally on accompanying documentation. Pseudocode, FSMs, and decision tables are typical means for describing SADT activity behavior. Douglass and Eickmeier use decision tables to specify the behavior of every lowest level activity.

Among other criticisms of SADT, Tse and Pong maintain that SADT is too dependent on functional breakdown, to complicated for users to fully understand, provides no smooth transition from specification to implementation and that it is too informal for automated verification (38:148).

2.3.2 Refine Refine is a formal specification environment and language that provides high level abstractions and operators. It supports execution of system specifications to examine specification accuracy and completeness.

Refine authors define the following goals for Refine (30:1-3):

1. Provide an integrated environment supporting high-level (abstract) programming.
2. Provide a tool to analyze and reformat (transform) programs.
3. Provide an extensible language— i.e., users can define domain specific programming environments. Allow users to define object classes, types, functions and grammars.

[7]Example entities are: a person, software routine, or hardware device.

4. Allow users to specify programs in whatever format or style they desire, including high level abstractions and low level procedural languages.
5. Support stepwise refinement— i.e., gradually convert specifications into implementations a step at a time (also supports "Rapid Prototyping").

Refine integrates set theory, logic, transformation rules, and pattern matching, providing a powerful programming environment including a parser and compiler (30:1-2). Refine's high level language constructs are not available in other commercially available languages (30:1-3). Refine stores programs, documents, test cases, etc. in a database called an *object base* or *knowledge base*. It provides the capability to implement program transformation and documentation systems (30:1-2).

The Refine compiler is a program transformation system which compiles refine programs into Lisp by applying program transformation rules defined in Refine itself (30:1-2).

Users can define their own languages by describing the grammar with BNF productions. Refine produces a lexical analyzer, parser, pattern matcher, pattern constructor and prettyprinter for user-defined languages (30:1-2).

2.3.3 VHDL Interestingly, hardware engineers have also been searching for a tool to specify the behavior of their systems. In 1981, the Department of Defense's VHSIC Program Office contracted for an industry standard and technology independent hardware description language which generates executable specifications. VHDL's goal is to describe systems at a number of different levels of abstraction, and to simulate systems at any mixture of those levels. (26, 12). Since the mid 1980's engineers have used VHDL to specify the behavior and structure of their designs, and they have used VHDL's event-driven simulator to validate them. In 1991, Eickmeier investigated using VHDL to specify and simulate software systems. We describe the features of VHDL that Eickmeier used, and the features that we use in our Research in the following paragraphs. Our focused discussion greatly simplifies VHDL, and we refer you to Lispett, Schaefer, and Ussery (26) and the IEEE Standard VHDL Language Reference Manual (19) for more detailed discussions of the language in general.

A *design entity* is the basic VHDL unit of description. A design entity is used to represent individual components or functions which make-up a system. VHDL allows users to generate multiple copies of an entity by instantiating it one or more times in a system. A design entity consists of an *Entity Declaration* and an *Architecture Body*. The Entity Declaration defines the inputs and outputs of the entity so other components can interface with it (12:12).

The SADT-like Figure 7 shows a Full-Adder entity. Other design entities interface with the Full-Adder design entity via the *Ports* `A`, `B`, `Carry_in`, `Sum`, and `Carry_out`.

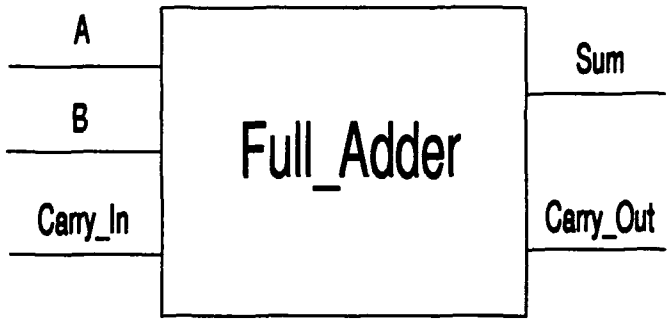

Figure 7. VHDL Full-Adder Entity Diagram (26, 12)

In VHDL code, the Full-Adder entity looks like Figure 8.

```
entity Full_Adder is
   port( A, B , Carry_In : in Bit; Sum, Carry_Out : out Bit);
end Full_Adder;
```

Figure 8. VHDL Full-Adder Entity Code

The *Architecture Body* describes the design entity in one of two ways (6, 26, 12):

- *structural description* A composition of existing design entities.

- *behavioral description* A procedural description of the entity's transformation of inputs to outputs.

A design entity can be described and decomposed using a whole hierarchy of structural definitions, but ultimately, lowest level entities have behavioral descriptions. Continuing with our Full_Adder example, suppose we have design entities for a Half_Adder and an OR_gate. We can use them to build our Full_Adder in a structural description as depicted in Figure 9. Alternatively, we could behaviorally describe the Full_Adder with an algorithm

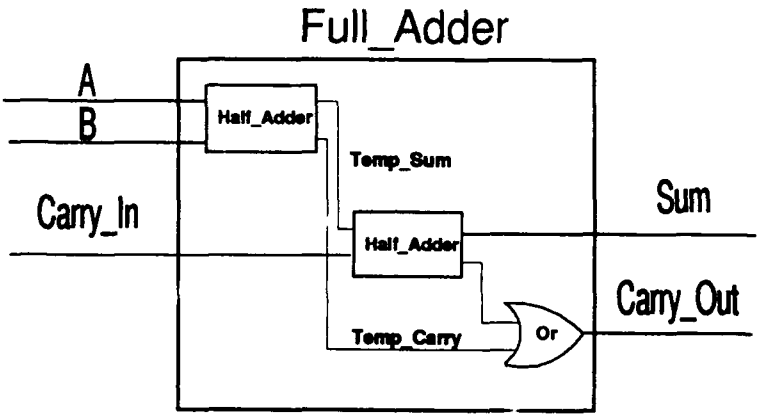

Figure 9. VHDL Full-Adder Architecture Body (26, 12)

that adds A, B, and Carry_In producing the Sum and Carry_Out. Our algorithm would be implemented in a *process.* A process is a sequential program, defined in an architecture body[8], which runs concurrently with all other processes during a simulation.

Entities communicate via *Signals.* Signals are like variables, except they are managed by *signal drivers.* A signal driver is like a queue; it holds the current value and all currently scheduled future values for the signal it is associated with. The simulator updates signals during event driven *simulation cycles.* Figure 10 depicts VHDL Simulation cycles. The simulator updates all signals according to values scheduled for the current simulation time

[8]Multiple processes can be defined in an architecture body.

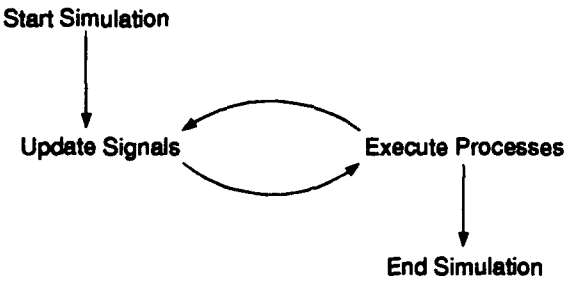

Figure 10. VHDL Simulation Cycle (26:12)

in the signal driver schedules, then it executes all processes that are sensitive[9] to the events[10], scheduling more events in the signal drivers. When there are no more signal events scheduled for the current time, the simulator increments the clock to the next event time. Repeated simulation cycles without the clock advancing are called *delta delay cycles.* The simulation cycle ends when no processes are sensitive to any updated signals. During the simulation, we access the clock through the predefined time valued variable *now.*

The signal drivers operate in conjunction with the VHDL simulator. Signal assignment examples in Figure 11 executed sequentially at time zero modify the signal driver queues resulting in signal values as shown in Table 1. Referring to Table 1, no signal

```
A <=  1;
B <=  2;
B <=  3 after 3 sec;
C <=  4, 5 after 3 sec, 6 after 5 sec;
C <=  transport 7 after 4 sec;
```

Figure 11. VHDL Integer Signal Assignment Example

assignment statement changes the signal value in the current simulation cycle; for example, the signal assignment to signal A does not take effect until the next delta delay. Note that the second signal assignment to signal B cancels the first signal assignment to B. This is what we call an *inertial* signal assignment, and the other type of signal assignment is *transport*, illustrated by the second signal assignment to signal C. Notice that the trans-

[9]Processes declare which signals they are sensitive to.
[10]VHDL events are signal value changes.

Table 1. VHDL Signal Assignment Values

Signal Name	Current Value	Next Delta Delay	1 sec	2 sec	3 sec	4 sec	5 sec	6 sec
A	0	1	1	1	1	1	1	1
B	0	0	0	0	3	3	3	3
C	0	4	4	4	5	7	7	7

port signal assignment does not cancel the first signal assignments to signal C, only the changes to C scheduled to occur after the second signal assignment delay (C never holds the value 6).

During the *Execute Processes* portion of the simulation cycle, processes are either *active* or *suspended.* A process is active when a event occurs on a signal in its *sensitivity list.* A sensitivity list is an optional part of a process declaration. The VHDL statement

```
Or_Process: process(A, B)
```

declares process Or_Process, and its sensitivity list includes signals A and B. A process is suspended when no events occur on signals in its sensitivity list during the current simulation cycle, or when no sensitivity list is defined and the process is waiting on a *wait statement* condition. Processes without sensitivity lists are allowed to have wait statements. There are four forms of wait statements[11] (26:67):

- *wait;* Permanently suspends a process.
- *wait on signal-list;* Similar to the process sensitivity list, the process remains suspended until an event on a signal in the signal-list.
- *wait until boolean-condition;* The process stays suspended until the boolean-condition is true.
- *wait for time-expression;* The process is suspended until time-expression time has passed.

[11]The different forms of wait statements can be combined— e.g., "wait on signal-list until boolean condition for time-expression;"

VHDL provides a series of signal attributes which we use to examine signal history information and to create new signals for process sensitivity lists[12]:

- *s'event* A boolean-valued attribute that detects a signal assignment on s during the current simulation cycle when the new signal value assigned is different from the previous signal value (26:265).
- *s'active* A boolean-valued attribute that detects a signal assignment on s during the current simulation cycle even though the new signal value assigned is the same as the previous signal value (a *transaction*[13]) (26:265).
- *s'last_event* A time-valued attribute that returns the amount of time elapsed since the last event on s (26:266).
- *s'last_value* An s-type-valûed attribute that returns the value of s before the last event on s (26:266).
- *s'stable(t)* A signal-valued attribute that generates a boolean signal with the value "true" if s has not had an event for t time otherwise, the new signal value is false (26:265).
- *s'transaction(t)* A signal-valued attribute that generates a bit signal whose value toggles (alternates between '0' and '1') every transaction (26:265).

2.3.4 Douglass and Eickmeier Conclusions Assessing his experience with Refine, Douglas concludes that Refine executable simulation is a major benefit when transforming SADT to Refine, pointing out improper behavior early in the development stage (10:86). He also states that the resulting Refine code serves as a formal foundation for the design and implementation phases of the software life cycle (10:87).

Eickmeier concludes that the SADT to VHDL transformation and VHDL simulation improves the requirements analysis process (12:147-148). Generating and executing the VHDL simulation validates the specification's completeness, consistency, feasibility, and

[12]This list includes only the attributes we use, not all attributes defined in VHDL.
[13]Every event is also a transaction.

testability (12:148). He states that the executable specification provides effective rapid-prototyping (12:148). Eickmeier's recommendations include substituting a state transition approach for behavioral description in place of decision tables and designing reusable VHDL components such as a linked list package.

Although both the heater and lift are real-time systems, Eickmeier and Douglass were not focusing on real-time system specification per se. But, among other things, they concluded that VHDL was better than Refine for specifying and examining timing requirements of both the heater and lift problems (10:77)(12:143). In fact, Eickmeier recommends a three step translation process (SADT to Refine to VHDL) because he believes it is easier to initially create the Refine implementation, iron out significant behavioral problems, and then move to VHDL to investigate the details of the behavior with respect to time (12:148).

2.4 Reacto

Kestrel Institute[14] is currently developing a product called Reacto, which works in conjunction with Refine to support software specification of reactive systems. Reacto's purpose is to "provide an environment that supports the acquisition and correct implementation of software specifications for reactive subsystems in the COMSEC [Communications Security] domain (23:2)." Reacto is based on hierarchical finite state machines similar to statecharts, but without concurrency mechanisms like orthogonal states (13:2). Users may specify a concurrent system by creating multiple FSMs communicating with shared interface variables. Reacto developers are focusing on the formal semantics of the FSM language and verification of state hierarchy assertions (13:2). Reacto has three subsystems, the *Reacto Editor*, *Reacto Compiler*, and the *Reacto Simulator*.

Users create formal specifications (R-Specs) with the Reacto Editor. Users compile, consistency check, and verify R-Specs using the Compiler subsystem, and they use the Simulator subsystem to execute and examine the behavior of the R-Spec.

2.4.1 Reacto Editor The Editor has three modes, *Graphics*, *Graphics-Outline* and *Developer*. In the Graphics and Graphics-Outline mode, users edit and create the Graphical

[14] Kestrel Institute is a research facility for Reasoning Systems.

structure of the FSM by specifying and naming hierarchical States and transitions between the states. These states and transitions become objects in the Refine knowledge base, and can be manipulated in all editor modes.

Collectively, the sets of state and transition objects associated with a particular state machine comprise the R-Spec (23:3,4). Once the state and transition objects are created, users save them from the knowledge base into text file, preserving the R-Spec in the Reacto spec library.

In the Editor's Developer mode, users examine and update the R-Spec text files via the public domain Emacs editor. Users can augment R-Spec states and transitions with global variables, Refine functions, and interface variables by creating an *auxiliary file* in the Developer mode (22:10).

In the Editor's Graphics-Outline mode, users can examine and update both R-Spec graphics and textual object attributes.

2.4.2 Reacto Compiler The Reacto Compiler's compile function transforms the R-Spec into target code[15] creating a fast load (.fasl) file which can be loaded later without recompiling. The Refine environment supports incremental parsing and compilation of the R-Specs.

The Reacto Compiler's consistency checking function checks to see if the R-Spec satisfies the syntactic and semantic constraints imposed by the system (23:3).

The Reacto Compiler's verifier function proves consistency of R-Spec behavior with regard to any state assertions users make (23:3). The verifier has three parts, a verification condition generator, a theorem prover, and a user interface. The theorem prover is based on natural deduction, term-rewriting, and hierarchical deduction (22:1). Users provide axioms to support the theorem prover in *Lemma files* (22:3). Users can incrementally apply the Verifier to States, transitions, or previously saved proof files (22:3-4). Users can interactively participate in the verification process, making inquiries and responding to forward and backward interaction prompts (22:13-14).

[15]Target code is Lisp code that is executable in the Refine environment.

2.4.3 Reacto Simulator The Reacto simulator is interactive and menu driven. During simulation, it highlights active states and transitions in the graphical display window depicting FSM behavior to users. It displays FSM output in a second window. The simulator prompts for input and takes input from users via the Emacs buffer. In a fourth window, the simulator displays the Reacto source code currently executing in conjunction with the active transition and states.

2.4.4 Reacto States Reacto States have the following user defined attributes:

- name
- substates
- initial-state
- own-vars
- assertion
- runtime-check

Users define a state's *name* via the graphics editor as they create the state.

The *Substates* attribute is the set of states hierarchically subordinate to the state. A state with no substates is a *primitive state*, and any state with substates is a *Superstate*. Users create substate attributes during Graphical or Graphics-Outline editing sessions (23:4).

An *initial-state* is the superstate's default substate. The default state may be another superstate or a substate. All superstates have an initial-state, and no primitive state has an initial-state. The first substate created during the Graphical or Graphics-Outline editing session is automatically the initial state of a superstate (23:4).

Own-vars is a set of variables defined for the state during subsequent Developer or Graphics-Outline editing sessions. These variables are visible in the state they are defined in and any substates of a superstate. Global variables and interface variables defined in an auxiliary file are visible in all states (23:4).

A state *assertion* is a boolean predicate over the state's visible variables. In the Developer editor configuration, users write state assertions on those variables to express the

properties of the specification that are applicable to the state. Assertions are the subject of the Verifier's proofs, and they can be checked during R-Spec execution as states are entered. Assertions are not necessary for execution, but they do give the user confidence that the specification is consistent (23:4). Superstate assertions are verified each time a substate of that superstate is entered. If an assertion fails during simulation, the user is notified during the simulation by an error message in one of the simulation windows. Assertions can be written to verify whatever the user wants to say about the state machine; Reacto is not limited to verification of system defined specification properties like STATEMATE.

The runtime-check attribute is a boolean predicate over the state's visible variables like the state assertion. The difference between the runtime-check and the state assertion is that runtime-checks are not subject to Verifier proofs (23:4). Like assertions, users add runtime-checks to the R-Spec during Developer or Graphics-Outline editing sessions.

2.4.5 Reacto Transitions The Reacto FSM model is a Mealy machine; all R-Spec behavior is associated with the transitions. Reacto transitions have the following user defined attributes:

- label
- from-state
- to-state
- history-flag
- predicate
- action
- priority

Users define a transition's *label* via the Graphics or Graphics-Outline Editor as users create the transition.

A transition's *from-state* is the originating state of the transition. When users define transitions from a superstate, the transition applies to all substates of that transition

except a *return*[16] state. The from-state is created during Graphics or Graphics-Outline editing sessions (23:5).

A Reacto transition's *to-state* is the destination state of the transition. When users define transitions to a superstate, the transition means that the FSM enters the default primitive substate of the to-state. History transitions return the FSM to the most recently visited primitive state of the to-state superstate (23:5). Reacto fills in the to-state attribute during Graphics or Graphics-Outline editing sessions.

The *history-flag* attribute is a boolean flag added by the user during Developer or Graphics-Outline editing sessions. If the history-flag is true, the transition must be a self loop on a single state (i.e., the from-state and to-state attributes are the same). Thus, when a history transition's from-state and to-state attributes are a superstate, it returns the FSM to the primitive substate of the superstate from which it started (23:5). Users can use history transitions to conveniently specify the same behavior for all sub-states of a superstate.

A transition's *predicate* is a boolean predicate over the visible variables which must be true for the transition to be taken (executed) (23:6). Users add the predicate during Developer or Graphics-Outline editing sessions.

Users create the transition *action* attribute during a Developer or Graphics-Outline editing session to define the behavior of the transition. Transition actions are the means for the FSM to accomplish its work. The work is accomplished by modifying the values of the originating states visible variables with assignment statements or function calls. Transition actions can also use the *call-state* mechanism to move the FSM to another superstate. States called via the call-state mechanism must be disconnected from the FSM, and control is returned to the originating state when a return state is entered (23:5-6). Since recursive call-states are not allowed, the call-state mechanism implements textual substitution (23:6), which is a means to specify the same behavior for the set of transitions calling the same state.

[16] A return state is a primitive state that originates no transitions.

Users define transition *priority* during Developer or Graphics-Outline editing sessions. Default priorities are zero. When more than one transition predicate is true for transitions of equal priority, the first one created (or loaded into the knowledge base from a saved R-Spec) is executed first (23:6).

2.4.6 Reacto I/O We do not use the predefined Reacto I/O facilities in our research. But, for the interested reader, we explain them in Appendix C.

2.4.7 Reacto Summary Reacto is so new, there are no published studies involving its use. It is a large system; the Refine/Reacto executable we are currently using is more than 35 megabytes of executable code and takes several minutes to load into memory on our SPARCstation 2.

Reacto comes with three example R-Specs, *Voting-Machine*, *KL-43*, and *Reference-Monitor*. The Voting-Machine is a simple R-Spec that authenticates voters, accepts and validates votes, and computes the tally (23:51). The KL-43 is the most sophisticated example R-Spec. It is an encrypting device designed to protect written communication over an unprotected channel (23:61). It demonstrates the use of the call-state mechanism. The Reference Monitor R-Spec is not explained in the *Reacto Users Manual*.

According to Gilham, Goldberg, and Wang, Reacto's goal is to provide a system that acquires and correctly implements real time SRSs (13:1). They claim Reacto implements formal FSMs with a convenient user interface and a verification system that checks the consistency between the specification and its operational behavior (13:7). Additionally, they assert that executable R-Specs provide rapid prototyping, guaranteed action termination, and data abstraction (13:8).

2.5 Summary

In this chapter, we discovered that specification languages for real-time systems should have the following properties:

- Abstract the real world well.

- Clearly understood by both specifier, implementer and user.
- Support verification that the specification and implementation are equivalent.
- Easy to modify and manipulate.
- Allow tracing of requirements.
- Executable specifications.
- Support specification of concurrency.
- Support specification of timing constraints.

It is difficult to achieve all of these properties in a single implementation. Refine and Reacto are perhaps the most abstract languages of those we've examined. STATEMATE and Reacto, because of their visual interface, are perhaps the most clearly understood. All of the languages except SADT support verification that the specification and implementation are equivalent because they are executable. They support running test cases and comparing the results of the test cases to the implementation. Some, like PAISLey, STATEMATE, and Reacto actually perform some verification of the specification itself. All of the languages purport relatively easy modification and implementation, and this is a somewhat subjective criteria. However, users of both PAISLey and SADT voice concern about the complexity of those languages which makes it more difficult to modify and manipulate specifications. The ability to trace requirements is supported best by STATEMATE's three views, and to a lesser extent by SADT's textual documentation. All of the languages except SADT provide executable specifications. The PAISLey, STATEMATE, SADT, and VHDL languages are the only languages that inherently support specifying concurrency at all levels in the specification. Reacto supports it by allowing separate specifications of communicating FSMs. PAISLey, STATMATE, and VHDL are the only languages that currently support concurrent executable simulations. To some extent, all languages can support specification of timing constraints, but only PAISLey, STATEMATE, and VHDL inherently provide features necessary to specify timing constraints and examine timing constraint consistency.

In support of our objectives, we keep these properties in mind, and we evaluate our methodology with them in Chapter VIII.

III. Improving Requirements Specification

Our stated objective is to investigate the feasibility, benefits, and problems associated with formalizing, validating, and verifying real-time SRSs using Reacto and VHDL.

In order to accomplish that objective we need three things. First, we need a methodology or process to organize and define the activities involved in validating and verifying real-time SRSs using Reacto and VHDL. Second, we need to define some significant supporting activities that must be accomplished before we can apply the methodology. And third, we need some example problems to illustrate the methodology. This chapter discusses each of these.

3.1 Methodology

Figure 12 summarizes the main focus of this research effort. Software engineers may recognize it as a data flow diagram. The "System Specification" store is the starting point. Initially, the store represents the informal system requirements as provided in a written English problem description. Arrows in Figure 12 represent data items that are

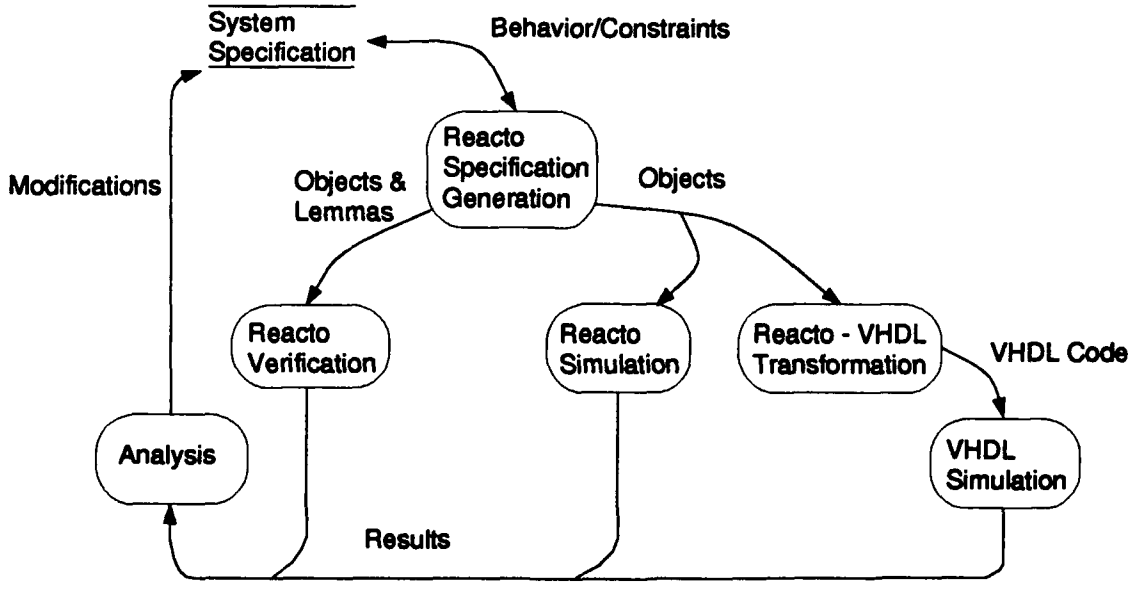

Figure 12. Reacto and VHDL Validation Process

passed among stores and activities. The boxes represent sub-activities or work units of the research effort.

From the System Specification store, we extract the behavior and constraints data that allow us to generate a Reacto state machine Specification (R-Spec). We organize behavior by creating a Reacto state machine that models the expected behavior. We organize constraints on the behavior in the Reacto state machine by assigning them to the applicable states and transitions. This formal R-Spec now takes the place of the informal written English specification in the System Specification store.

Once we have generated an R-Spec we take one of three paths. The first path we can follow leads to the Reacto Verification activity. This activity consists of using Reacto's automated verification system to formally verify the R-Spec which at this point is a set of objects in Refine's knowledge base. In addition to state machine objects, we can provide a lemma file (a file containing axioms) when required for certain proofs. We subsequently analyze verification results, and if necessary make modifications to the System Specification, which is now the formal R-Spec.

The second path out of the Reacto Specification Generation activity leads to the Reacto Simulation activity. The simulation activity is exercising the R-Spec interactively by updating inputs to the state machine manually, or by inputting a simulation file. We subsequently analyze simulation results, and if necessary make modifications to the System Specification.

The third path out of the Reacto Specification Generation activity leads to the Reacto-VHDL Transformation activity. This is the activity where the R-Spec objects are used to generate a formal VHDL state machine Specification (V-Spec). In Chapter V, we describe this activity in detail. For the purposes of this research effort we make this transformation manually using the R-Spec source files as input, and outputting VHDL source files. An automated transformation of the R-Spec in Refine's knowledge base to the V-Spec source file is possible, and we discuss the feasibility of an automated generation in Section 5.8.

Once the transformation is complete, and in preparation for the VHDL simulation, we manually configure the V-Spec or manually generate a VHDL testbench and test configuration. Once configured, we simulate the V-Spec using the VHDL debugger or using the testbench. Testbench simulations are similar to our Reacto file driven simulations. We subsequently analyze verification results, and if necessary make modifications to the System Specification, which is now the formal R-Spec, starting the Reacto and VHDL Validation Process over.

3.2 Supporting Activities

While Figure 12 represents the main focus of this research effort, there are significant supporting activities which are necessary to implement it. Getting to the point where we can apply Reacto and VHDL to real-time SRSs requires some additions to Reacto to manage time, and definition of the transformation process from Reacto to VHDL.

Reacto allows us to improve on traditional state diagrams using hierarchical state machines. With hierarchical state machines, we can specify complex behavior more efficiently and more understandably. But we must describe time and time constraints, something which is not currently provided in Reacto itself.

We define a method for augmenting R-Specs with time. This includes specifying time constraints, implementing a system clock, transition delays, timers, and making assertions about time. We illustrate using Reacto to specify real-time systems by applying it to two problems. Finally, we annotate the specification improvements that are made as a result of using Reacto.

Once we are satisfied with the Reacto verification and simulation results, we specify, simulate and validate the R-Spec in VHDL, and annotate the specification improvements made as a result of the VHDL simulations.

Before we can do this, we must transform the R-Spec into a V-Spec and verify that the Reacto state machine and VHDL state machine display equivalent behavior. Reacto

primitive states and transitions map directly to VHDL states and transitions[1]. Although the two state machines are basically the same, Reacto and VHDL themselves are significantly different.

Reacto is a more abstract language; for example, it allows the use of sets and sequences, providing quantification on those sets and sequences without concern for the details of implementation. VHDL, on the other hand, is a procedural language, with no predefined sets, sequences or quantification operators. Reconciling this difference between the languages is not trivial, especially when we look forward to an automated transformation. For example, implementing sets in VHDL requires generating a set data type and the traditional set operators like union, intersection, set membership, etc.; additionally, because we need to support quantification it requires generating VHDL functions that perform the quantification operation and return boolean values. Since VHDL is a strongly typed language, providing these general set operations is difficult.

The fact that VHDL is based on time and Reacto is not is another significant difference between the two languages. For example, we must manage a system clock in Reacto with the Reacto state machine, in VHDL, the simulator manages the clock In VHDL, the simulator schedules transition execution, in Reacto, transitions update the clock leading to a strictly sequential simulation.

Not only are Reacto simulations sequential, but Reacto is a sequential language. In contrast, VHDL is a concurrent language. While VHDL typically runs on sequential processors, it implements concurrency via an event driven simulation as described in Section 2.3.3. We exploit VHDL's concurrency to increase simulation power by simulating multiple state machines instead of a single state machine at a time like in Reacto. VHDL's concurrency allows us to independently update input to those state machines regardless of the state machine transitions.

Because of the differences between Reacto and VHDL, the translation between them is not completely straight forward. But as we exploit their differences, we benefit. Exploiting the differences between Reacto and VHDL helps us to understand the informal

[1]We do not use Reacto interface variables or call states, hence we provide no mapping between Reacto and VHDL for these two Reacto objects.

problem quickly and produce a better SRS. Another benefit is increased understanding of real-time system specification in general, especially with concern for time constraints. Finally, it leads us to discover ways we can improve Reacto and VHDL and our use of them.

3.3 Example Problems— Introduction

To illustrate the use of our methodology, we examine two relatively small real-time problems. The first problem is a typical cruise control, and the second, a more complicated lift (elevator) control system.

3.3.1 A Cruise Control The Cruise Control problem is a portion of the automobile Management System problem[2] from Derek J. Hatley and Imatiaz A. Pirbhai's book *Strategies For Real-Time System Specification* (17).

We have several reasons for choosing the cruise control problem. First, cruise control is a well known and simple example of a real-time problem with other published solutions for comparison (34, 17). The case study in Hatley's book includes well defined response-response timing constraints, with a rate constraint (call it a form of response-response timing constraint). We have a Cruise-Control specification developed using Eickmeier's methodology, and are familiar with the problem.

3.3.1.1 Cruise Control Problem Statement The cruise control function is to take over the task of maintaining a constant speed when commanded to do so by the driver.

3.3.1.2 Cruise Control Requirements The driver must be able to enter several commands (*R1*), including: Activate, Deactivate, Start Accelerating, Stop Accelerating, and Resume. The cruise control function can be operated any time the engine is running and the transmission is in top gear (*R2*). When the driver presses Activate, the system selects the current speed, but only if it is at least 30 miles per hour, and holds the car at

[2]The Automobile Management System problem was derived from a problem used at the Department of Defense's July 1985 Software Technology for Adaptable, Reliable Systems Methodology Conference in Colorado Springs, Colorado.

that speed (*R3*). Deactivate returns control to the driver regardless of any other commands (*R4*). Start Accelerating causes the system to accelerate the car at a comfortable rate until Stop Accelerating occurs, when the system holds the car at this new speed (*R5*). Resume causes the system to return the car to the speed selected prior to braking or gear shifting (*R6*).

The driver must be able to increase the speed at any time by depressing the accelerator pedal (*R7*), or reduce the speed by depressing the brake pedal (*R8*). Thus, the driver may go faster than the cruise control setting simply by depressing the accelerator pedal far enough. When the pedal is released, the system regains control (*R9*). Any time the brake pedal is depressed, or the transmission shifts out of top gear, the system must go inactive (*R10*). Following this, when the brake is released, the transmission is back in top gear, and Resume is pressed, the system returns the car to the previously selected speed (*R11*). However, if a Deactivate has occurred in the intervening time, Resume does nothing (*R12*).

3.3.1.3 Cruise Control Constraints The system controls car speed by driving a throttle actuator. The actuator operates in parallel with the gas pedal mechanism. Whichever demands greater speed controls the throttle. The cruise control drives the throttle actuator with an electrical signal that varies linearly in accordance with throttle deflection. It ranges from 0.0 volts to 8.0 volts (*C1*); 0.0 volts closes the throttle, and 8.0 volts sets the throttle wide open.

When the cruise control senses speed more than 2 mph above the selected speed, it closes the throttle completely (*C2*)— e.g. when coasting down a steep hill. At speeds below this, it is to drive the throttle to a deflection proportional to the speed error until, at 2 mph below the selected speed, the throttle is wide open (*C3*)— e.g. when climbing a steep hill. Thus, the cruise control is the feedback or control part of a servo loop, in which the engine is the feed-forward part. To operate smoothly, the system must update its outputs at least once per second (*C4*).

To avoid uncomfortable acceleration, the actuator must not open faster than 0.8 volts per second (*C5*). It may close at any rate however, since the car just coasts when the throttle is closed. According to automotive engineers, these characteristics keep the car

within 1 mph of the selected speed on normal gradients giving a smooth and comfortable ride.

When the system is accelerating the car, it must measure the acceleration and hold it at 1 mph/sec. Again the gradient affects the throttle setting. When acceleration reaches 1.2 mph/sec, the system should close the throttle (*C6*); at 0.8 mph/sec, it should set it wide open (*C7*). Between these limits, the throttle setting should vary linearly with acceleration (*C8*).

We summarize cruise control timing constraints in Table 2.

Table 2. Cruise Control Timing Constraints (17:286)

Input		Output	
Signal	Event	Value	Response Time
Activate	goes true	cruise*	500 ms
Resume	goes true	cruise*	500 ms
Deactivate	goes true	zero	500 ms
In-Top-Gear	goes false	zero	500 ms
Braking	goes true	zero	500 ms
Start-Acceleration	goes true	accelerate*	500 ms
Stop-Acceleration	goes true	cruise*	500 ms
Engine-Running	goes false	zero	500 ms
Current-Speed	changes	changes*	1000 ms
Acceleration	changes	changes*	1000 ms
* Voltage Change $\leq$ 0.8 Volts per second			

3.3.1.4 Cruise Control Environment We use SADT ala Eickmeier (12) to enhance the visualization of the R-Spec. The state machine description does not overtly show us what the inputs and outputs of the state machine are. As Zave and Jackson point out, we must express this information in a form that can be easily understood by users and developers (42:87). The SADT diagram in Figure 13 depicts the cruise control in its environment.

3.3.2 A Lift Controller System We have several reasons for choosing the lift controller system as a second example to illustrate our methodology. First, the lift system

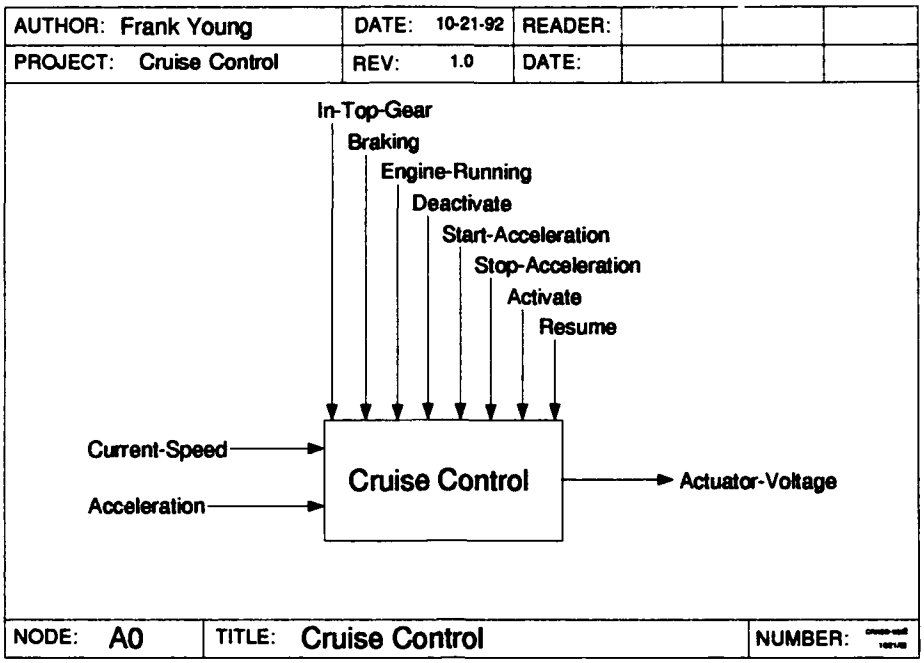

Figure 13. Cruise Control in the Environment

problem is larger than the cruise control. It tests the ability of our method to accommodate a larger design, and our ability to model instantiations of an arbitrary number of identical objects. Recall from Section 2.2.2 that one of the problems associated with STATEMATE is its inability to deal with multiple instantiations of identical state machines. In our example we deal with lifts as opposed to bank teller machines (34:54). Additional types of timing constraints is the second reason for choosing the lift controller problem. In addition to the types of timing constraints in the cruise control problem, the lift controller includes response-response and average-response-time timing constraints. It also includes both a minimum and maximum stimulus-response time constraint. Another reason we use the lift controller system, is that we have have more than one study to compare with; for example, Douglas (10), Eickmeier (12), and Spicer (36) implemented simulations of lift control systems here at AFIT. Yourdon (40) and Wood (39) include published lift controller studies.

3.3.2.1 Lift Controller Problem Statement The Lift controller schedules and controls one or more lifts in a multi-story building. We use the basic Lift Control System problem statement from Eickmeier: (12:82-83).

1. Each lift has a set of buttons, one for each floor. These illuminate when pressed and cause the lift to visit the corresponding floor. The illumination is cancelled when the corresponding floor is visited by the lift.
2. Each floor has two buttons (except ground and top floor), one to request an up-lift and one to request a down-lift. These buttons illuminate when pressed. The illumination is cancelled when a lift visits a floor and is either moving in the desired direction, or has no outstanding requests. In the latter case, if both floor request buttons are pressed, only one should be cancelled. The algorithm to decide which to service first should minimize the waiting time for both requests.
3. When a lift has no requests to service, it should remain at its final destination with its doors closed and await further requests....
4. All requests for lifts from floors must be serviced eventually, with all floors given equal priority....
5. All requests for floors within lifts must be serviced eventually, with floors being serviced sequentially in the direction of travel....
6. Each lift has an emergency button which, when pressed causes a warning signal to be sent to the site manager. The lift is then deemed 'out of service'. Each lift has a mechanism to cancel its 'out of service' status (12:83).

We augment Eickmeier's problem description with two behavioral requirements from Yourdon's elevator problem, since it is our intention to specify a lift controller, not a lift system (i.e., lift controller, lifts, doors, motors...).

1. Each lift is connected to a floor sensor. The floor sensor indicates which floor the lift is at. It also indicates when the lift is between floors (40:633).
2. Each lift is connected to a motor. The motor can move the lift up, down or stop the lift (40:633-634).

As discussed in Yourdon, we assume the lift mechanism does not obey unsafe commands (40:634). Specifically, we don't worry about stopping the lift exactly at floor

level, or making sure doors are closed before we send a command to turn on the motor. We assume that the manufacturer's electromechanical door and motor controls are interfaced such that the safety requirements are met. For example, if the lift controller turns on the lift motor while the doors are open, the lift will not actually move until the doors are closed.

Timing constraints are missing from Eickmeier's specification. We augment the lift specification with the following timing constraints; some are adapted from March's Thesis (27), and we generate others to illustrate the different types of constraints.

1. The lift controller must turn on destination lights and summons lights within 100 milliseconds after a destination or summons button is pressed (27:A-25).
2. The lift controller must turn off destination lights and summons lights within 100 milliseconds after a summons or destination request is satisfied (27:A-24-A-25).
3. A lift must wait between 3 and 5 seconds after it turns the motor off before it turns the motor on again (27:A-25).
4. After a lift has been summoned to a floor, the lift must wait at least 10 seconds before it changes directions when no destination buttons are pressed.
5. The lift controller must stop the lift within 100 milliseconds after the lift's emergency button is pressed when the lift is at any floor.
6. The average response time to summons requests should be less than 20 seconds (27:A-24).

We summarize lift controller timing constraints in Table 3.

3.3.2.2 Lift Controller Environment Like the cruise control, we use SADT to enhance the visualization of the lift controller R-Spec. The SADT diagram in Figure 14 depicts the lift control system in its environment. The terms *Dest-Buttons* and *Dest-Lights* designate the bu s/lights internal to the lift. The terms *Up-Buttons, Down-Buttons* and *Up-Lights, Down-Lights* designate the summons buttons and lights on each floor.

Table 3. Lift Controller Timing Constraints

Constraint	Response Time
Light-On-Time-Limit	100 ms
Light-Off-Time-Limit	100 ms
Min-Wait-Time	3 sec
Wait-Time-Limit	5 sec
Timeout-Timer-Duration	10 sec
Emergency-Stop-Time-Limit	100 ms
Average-Response-Limit	20 sec

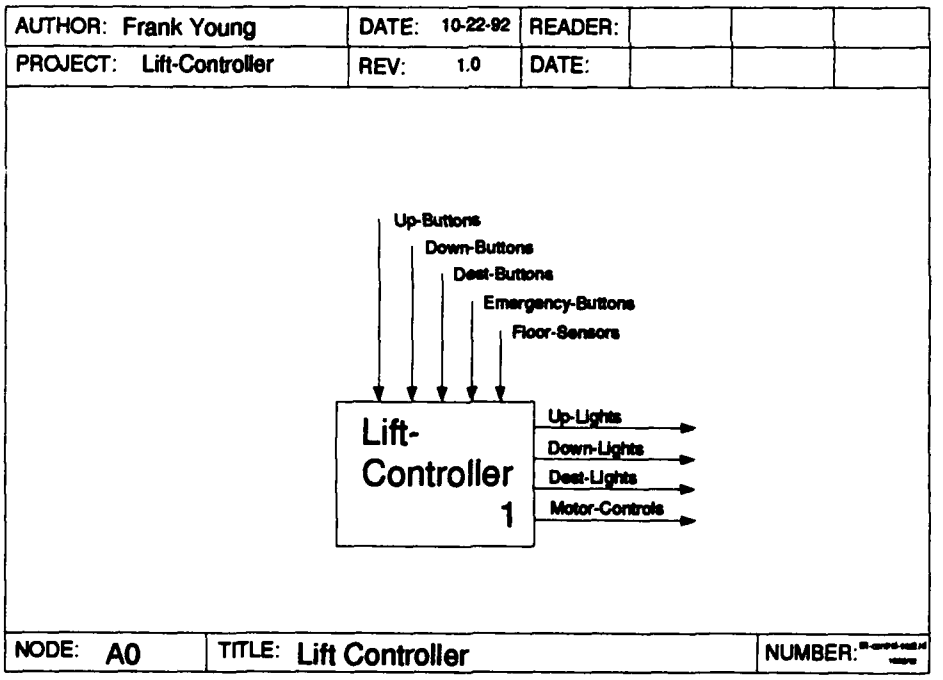

Figure 14. Lift Control in the Environment

In order to illustrate the power of our methodology to handle an arbitrary number of identical objects, we break the lift controller down into two activities, schedule-lifts and lift. When we want to specify an architecture connecting two or more state machines or other activities together, we use SADT to specify the connecting details. The SADT diagram in Figure 15 depicts the lift control system broken down into these activities. In

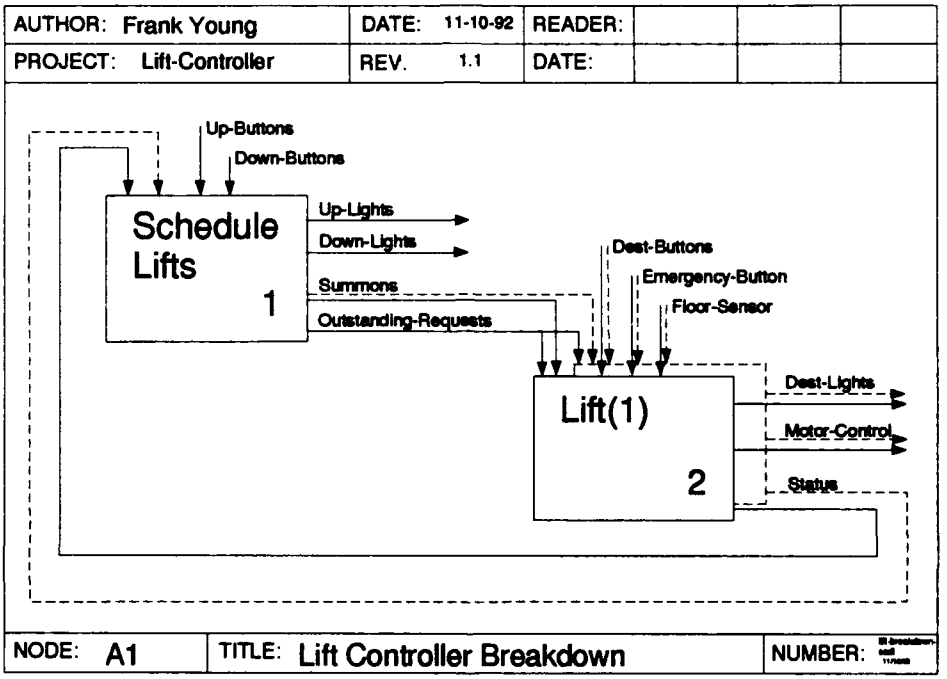

Figure 15. Lift Controller Activities

Figure 15 we represent multiple lift activities by showing the inputs and outputs of two lifts. We show the inputs and outputs for the second lift as dashed lines in Figure 15. Each lift object is identical, and the number of lifts is not limited to two.

In Figure 15 *Summons* represents a unique set of summons requests for each lift and *Outstanding-Requests* is the set of summons requests for all lifts.

3.4 Summary

Two significant supporting activities we must accomplish to fulfill our research objectives are:

- Augment Reacto with timing.

- Define Reacto to VHDL mapping and translation process.

In support of our objective we define a methodology incorporating the following steps:

1. Augment Reacto with timing.
2. Generate Formal R-Spec from informal requirements.
3. Verify R-Specs with Reacto Verifier, make R-Spec corrections.
4. Simulate R-Specs, make R-Spec corrections.
5. Define Reacto to VHDL mapping and translation process.
6. Apply transformation to convert R-Specs to V-Specs.
7. Simulate V-Specs, examine timing in detail, make R-Spec and V-Spec corrections.

We have two sample problems that we will use to demonstrate our methodology, first, a fairly simple cruise control, and second, a more complex lift controller system.

IV. Modeling Time and Applying Reacto

Our stated objective is to investigate the feasibility, benefits and problems associated with formalizing, validating, and verifying real-time SRSs using Reacto and VHDL. In Chapter III we defined a methodology for accomplishing that objective, identified some significant supporting activities, and introduced two example problems to illustrate the methodology. In this chapter we narrate accomplishing that objective by describing the *Modeling Timing Requirements in Reacto* supporting activity and applying Reacto to the the two example problems. We begin by describing the modeling-time-in-Reacto supporting activity.

4.1 Modeling Timing Requirements in Reacto

There are certain augmentations that are absolutely necessary to successfully model temporal behavior in Reacto and other augmentations which could improve our ability to model it. First, we describe the augmentations we implement in Section 4.1.1, then we show a very simple example in Section 4.1.2. Finally, we discuss augmentations to consider for implementation in Section 4.1.3.

4.1.1 Reacto Augmentations to Implement In order to model and measure time, we define a means to track and quantify time by declaring a global variable, ***clock***, of type integer. We call one time unit *simulation granularity.* For the purpose of our cruise control and lift systems we establish a simulation granularity of one millisecond because that is the smallest unit of time we want to specify. To model a microprocessor, we might use a nanosecond, femtosecond, or picosecond; to model the solar system we might choose a day, month, year, or century. The point is, we choose a small enough time value to allow meaningful examination of system behavior.

Along with clock, we need to express time constraints. We declare constants of type integer in the auxiliary file to represent constraints. We call maximum stimulus-response constraints *time limits* and minimum stimulus-response constraints *min times.* We call response-response constraints *durations.*

Additionally, we need some means to express the amount of time work takes. We define work as follows: If X is an input to our FSM, and the output is some function of X, i.e. Y(X), *work* is the effort required to produce Y(X). And, the *time work takes* is the delay between X changing and the responding change in Y(X)[1]. We use *transition delay* to express the amount of time the work assigned to a transition takes. Like the clock and constraints, we declare transition delays as constants of type integer in the auxiliary file. We increment the clock by the transition delay as the transition executes to model time passing.

Next, we define a means to validate that the FSM behavior does in fact meet the stimulus-response time constraints. We could use error states and error transitions to the error states to verify FSM behavior as described in Section 2.1.2, but by using Reacto assertions, we can discover inappropriate temporal behavior without the error states. To do so, we need a means to mark the time that events occur. We mark the time that events occur with *start time logs.* We declare start time logs as variables of type integer in the auxiliary file. We use start time logs to measure stimulus-response time constraints. Initially, we *reset* start time logs to either Max-Time-Reset or Min-Time-Reset[2] depending on whether we use them to verify a maximum or minimum time stimulus-response constraint. A timing constraint with both a minimum and maximum constraint requires separate timers, since timer reset values are different for minimum and maximum constraint timers. A transition uses start time logs to log the time when it starts responding to an event. As the transition executes, it updates the clock by its delay. After the transition moves the FSM to the next state, Reacto executes the state assertion and notifies us if the transition failed to meet the time constraint. Figure 16 shows the general form of a Reacto assertion to check a stimulus-response time constraint (on X) with both a minimum and maximum constraint. In Figure 16, Min-X-Start-Time and Max-X-Start-Time are start time log variables, and Min-X-Time and X-Time-Limit are the minimum and maximum constraint constants.

Next, as discussed in Section 2.1.2, we need some means to model response-response timing constraints. Response-response constraints always generate a transition whose pred-

[1]If Y(X) is not a bijection, the responding change may be the same as the previous value of Y(X).
[2]Max-Time-Reset is a very large integer, and Min-Time-Reset is a very small (negative) integer.

```
assertion Clock - Min-X-Start-Time >= Min-X-Time &
          Clock - Max-X-Start-Time <= X-Time-Limit
```

Figure 16. Reacto Timing Constraint Assertion

icate involves a time value (see Figure 17). We use *timers* and durations to model response-response time constraints. Timers are the means by which we measure the time between response-response events. Similar to start time logs, we declare timers as variables of type integer in the auxiliary file and initialize them to Max-Time-Reset. As transitions perform behavior subject to a response-response constraint, they set timers. Reacto examines transition predicates of the form illustrated in Figure 17 (sensitive to response-response constraint X) and executes the response-response sensitive transition when the predicate is true. In Figure 17, X-Timer-Duration is X's response-response time constraint constant, and X-Timer is the timer variable.

Specifiers must consider when to reset start time logs and timers. Generally, they should be reset when a transition is taken from a state after they are used in a state assertion or used to meet a response-response time constraint, but they may apply to a group of states or a group of transitions accomplishing some constrained behavior, and should remain set for two or more transitions. We reset timers and logs with Max-Time-Reset or Min-Time-Reset to indicate that we are finished evaluating whether or not the constraint is met, or we can reset them with the current clock value if the currently executing transition is also subject to the same time constraint. It ultimately depends on the behavior we are trying to specify.

Have we described all we need to model the timing constraints of systems? We can specify stimulus-response and response-response timing constraints. But can we model

```
predicate Clock - X-Timer-Duration >= X-Timer
```

Figure 17. Reacto Timer Sensitive Predicate

them correctly? Specifically, does our Reacto model support modeling systems with response-response constraints correctly? If we specify a response-response constraint, and our system reaches a steady state (no transition predicates are true) after we have set the timer, how does time advance to the time when the timer goes off[3]? To solve this problem, we add one more transition to every FSM. The *wait* transition as shown in Figure 18, is the lowest priority transition, and it is a history transition effective in all states. Wait's predicate is "true" so it always executes when no other transition predicate evaluates to "true". As the lowest priority transition, wait executes only when no other transition predicates

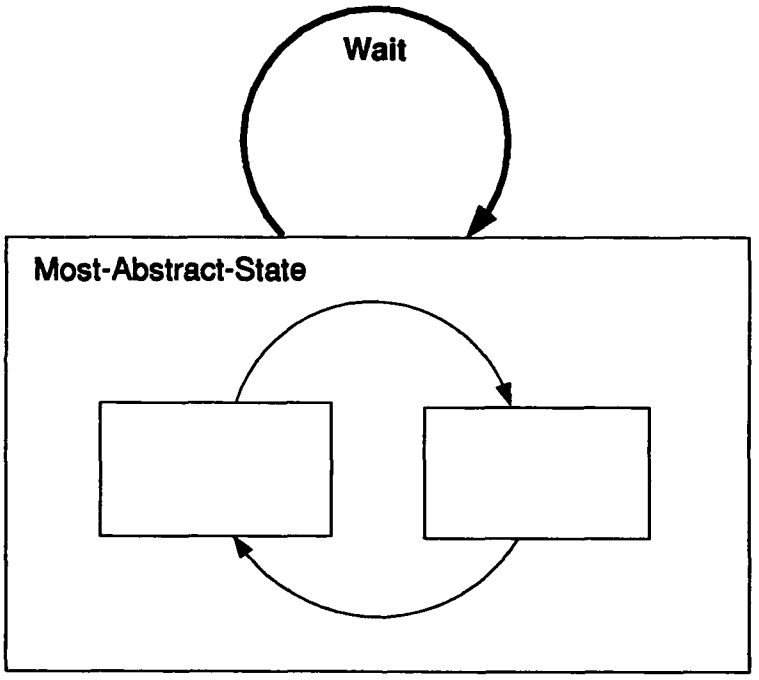

Figure 18. Reacto Wait Transition

are true. Wait's functions are to get input and update the system clock. Additionally, it resets timers as specified by the engineer to avoid time constraint assertion errors for assertions that have already been verified. We set the constant Wait-Time-Step to establish how much the wait transition updates the clock as it runs. We determine Wait-Time-Step based on the requirements we have for time passing, usually it is some even divisor of the response-response timing constraints.

[3]Remember— Transition execution is the only means to increment the clock.

This completes our definition of temporal augmentations we implement in Reacto. We review them by examining a simple example.

4.1.9 Relay Augmentation Example Suppose we are modeling a relay switch, with two states On and Off, as depicted in Figure 19.

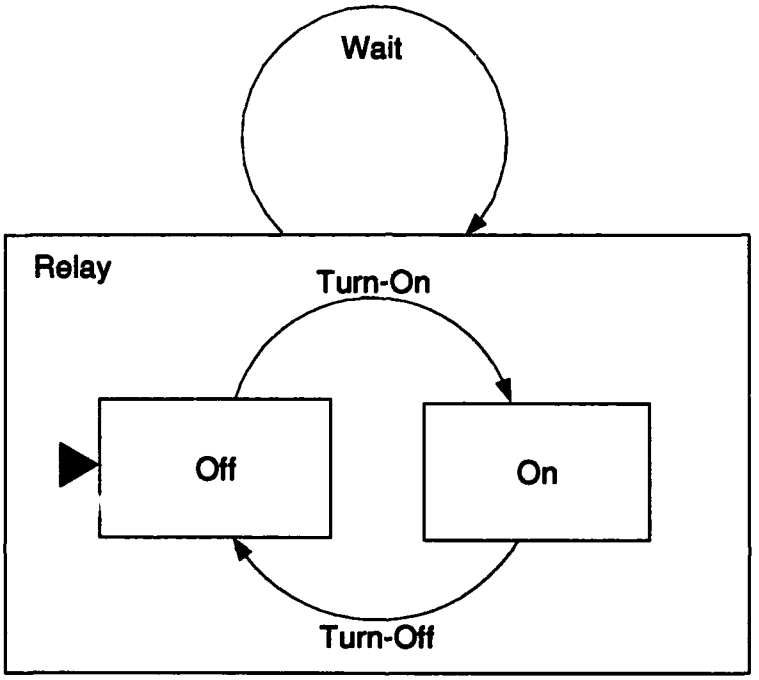

Figure 19. Reacto Relay FSM

Assume that the relay has two inputs, an On-Button and an Off-Button, and a single output Power, all of which we model as booleans initialized to false. Also assume that the switch is constrained to turn on or turn off within 1 second of an On-Button or Off-Button press respectively. So, we have two stimulus-response constraints, On-Time-Limit and Off-Time-Limit, both set to 1 second. We declare two start time logs, On-Start-Time and Off-Start-Time to log On-Button and Off-Button event times. We set Turn-On and Turn-Off delays to their constraint values, 1 second. The assertions, predicates and actions of our FSM are shown in Figure 20. Let us walk through a scenario, turning the relay on and off as depicted in Figure 21, a scenario timing diagram. Initially, time is 0 seconds, the relay is in the Off state, and both buttons are false. The Relay FSM is in a steady state, no

```
Off assertion : Power = false and
                Clock - Off-Start-Time <= Off-Time-Limit
On assertion  : Power = True and
                Clock - On-Start-Time <= On-Time-Limit

Turn-Off predicate : Off-Button = true
         action    : On-Start-Time <- Max-Time-Reset;
                     Off-Start-Time <- Clock;
                     Clock <- Clock + Turn-Off-Delay;
                     Power <- false;
                     get-input

Turn-On  predicate : On-Button = true
         action    : Off-Start-Time <- Max-Time-Reset;
                     On-Start-Time <- Clock;
                     Clock <- Clock + Turn-On-Delay;
                     Power <- true;
                     get-input
```

Figure 20. Reacto Relay Assertions, Predicates, and Actions

transition predicates are true except for the wait transition, so Wait executes. First, Wait resets the two start time logs On-Time-Limit and Off-Time-Limit to Max-Time-Reset. Second, Wait updates the time from 0 seconds to 1 second. Finally, Wait provides an opportunity to input events. The get-input function tells us that at time 1 second, Power is false, so we set On-Button to true. As Wait finishes executing, Reacto evaluates the Off assertion, which is true.

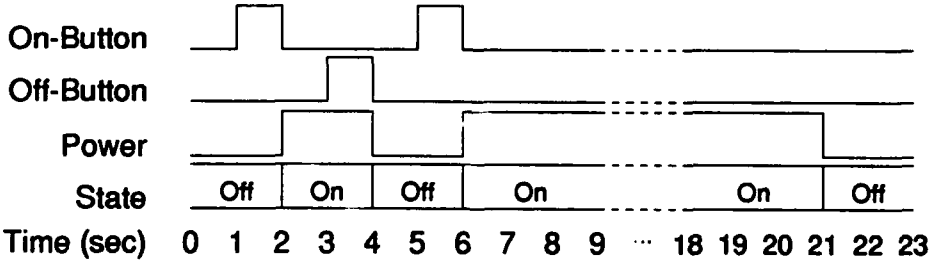

Figure 21. Relay Scenario Timing Diagram

Now, the time is 1 second, and Turn-On predicate is true, so transition Turn-On is executed. First, Turn-On resets Off-Start-Time to Max-Time-Reset, and On-Start-Time to Clock (currently Clock is 1 second). Next it increments the Clock adding its delay, so the time is now 2 seconds. Then it turns on the relay by setting Power true. Finally, it calls get-input which tells us that Power is now true, and we set On-Button to false. Now, as Turn-On finishes executing, Reacto evaluates the On assertion, which is true.

Once again, the only predicate that is true is Wait's. Wait resets the start time logs, increments the Clock to 3 seconds, and this time, we set Off-Button true. As Wait finishes executing, Reacto reevaluates the On assertion, which is true.

Now, the time is 3 seconds, Turn-Off's predicate is true, and Turn-Off is executed. First, Turn-Off resets On-Start-Time to Max-Time-Reset, and Off-Start-Time to Clock (currently Clock is 3 seconds). Next it increments the Clock adding its delay, so the time is now 4 seconds. Then it turns off the relay by setting Power false. Finally, it calls get-input which tells us that Power is now false, and we set Off-Button to false. Now, as Turn-On finishes executing, Reacto evaluates the Off assertion, which is true.

Let's complicate our example by adding a response-response constraint. Assume that the equipment controlled by our relay cannot be on for more than 15 seconds. This means that Relay must turn itself off automatically 15 seconds after it turns on. We add a timer *On-Timer* initialized to Max-Time-Reset, and specify the response-response constraint in the constant *On-Timer-Duration*, setting it to 15. We modify Turn-Off's predicate to read:

```
Off-Button = true or Clock - On-Timer-Duration >= On-Timer
```

Now, time is 4 seconds, the relay is in the Off state, and both buttons are false. The Relay FSM is in a steady state, no transition predicates are true except for the wait transition, so Wait executes. First, Wait resets the two start time logs On-Time-Limit and Off-Time-Limit to Max-Time-Reset. Second, Wait updates the time from 4 seconds to 5 seconds. Finally, Wait provides an opportunity to input events. The get-input function tells us that at time 5 seconds, Power is false, so we set On-Button to true. As Wait finishes executing, Reacto evaluates the Off assertion, which is true.

Now, the time is 5 seconds, and Turn-On predicate is true, so transition Turn-On is executed. First, Turn-On resets Off-Start-Time to Max-Time-Reset, and sets both On-Start-Time and On-Timer to Clock (currently Clock is 5 seconds). Next it increments the Clock adding its delay, so the time is now 6 seconds. Then it turns on the relay by setting Power true. Finally, it calls get-input which tells us that Power is true at 6 seconds, and we set On-Button to false. Now, as Turn-On finishes executing, Reacto evaluates the On assertion, which is true.

Once again, the only predicate that is true is Wait's. Wait resets the start time logs, increments the Clock to 7 seconds, and this time, we input no events. As Wait finishes executing, Reacto reevaluates the On assertion, which is true. As long as we input no more events, Wait executes repeatedly, telling us the current time and that Power is true, giving us the opportunity to input more events, and adding 1 second to the clock. Each time, Reacto evaluates the On-State assertion which is always true. Finally, when Clock equals 19 seconds, wait executes one more time incrementing the Clock to 20 seconds.

Now, Turn-Off's new predicate, Clock − On-Timer-Duration $\geq$ On-Timer, is true ($20 - 15 >= 5$) and Turn-Off is executed. First, Turn-Off resets On-Start-Time and On-Timer to Max-Time-Reset, and Off-Start-Time to Clock (currently Clock is 20 seconds). Next it increments the Clock adding its delay, so the time is now 21 seconds. Then it turns off the relay by setting Power false. Finally, it calls get-input which tells us that Power is false at 21 seconds. Now, as Turn-Off finishes executing, Reacto evaluates the Off assertion, which is true.

This completes our definition of temporal augmentations we implement in Reacto, and a review of them via a simple example. Now, let us discuss augmentations we do not implement and justify our decision not to implement them.

4.1.3 Reacto Augmentations to Consider In the previous section we define all of the time extensions we use to model our sample problems, and provide a simple example using them, but we know that there are more augmentations possible to improve modeling time constraints in Reacto.

For example, updating the clock by the transition delay prohibits us from modeling events occurring in the time interval between transition a's update of the clock, *t1*, and the transition b's update of the clock, *t2*. Figure 22 illustrates this problem. We call events

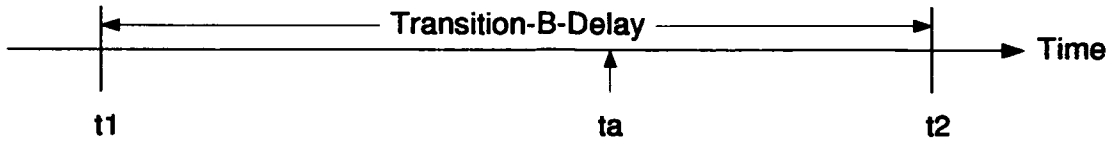

Figure 22. Asynchronous Event

occurring between times *t1* and *t2* (e.g., event A occurring at time *ta*) *asynchronous events.* When we model event A, we must inject it into the simulation at either t1 or t2. Therefore, our methodology constrains us to model events at times dictated by transition delays.

One way to model asynchronous events in Reacto is to specify intermediate states between actual states, and specify a *New-Wait* transition with an arbitrarily short delay to pass the time between those states as depicted in Figure 23. This solution requires tran-

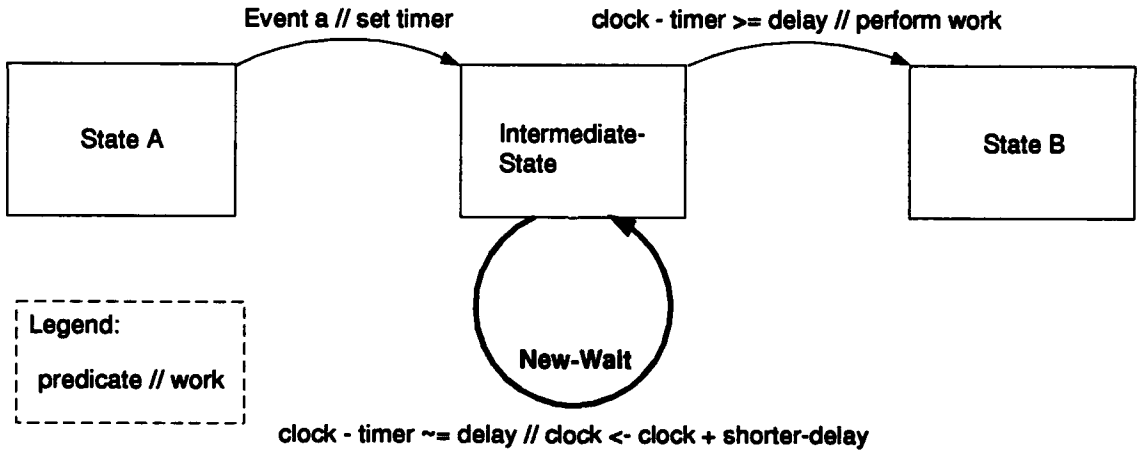

Figure 23. Reacto Asynchronous Event Solution

sitions from the intermediate states to all of the other states (actually other intermediate states) which are reachable from state A, exponentially exploding the number of states and transitions. Those intermediate states and extra transitions are not intuitively part of the SRS, but create an R-Spec specifically tailored to solve the asynchronous events problem within our Reacto methodology. The intermediate states and extra transitions greatly complicate the spec and reduce our ability to understand and manage it.

Another way to fix this asynchronous event problem is to create a true event driven simulation. Since Reacto currently limits us to expressing behavior via transition actions only, such a task is complicated. It requires some means to schedule and cancel events, as well as a more sophisticated clock than we have currently described. We have two reasons for not attempting to implement event driven simulation in Reacto. First, we assert that implementing an event driven simulation could be accomplished much easier by modifying the Reacto simulator than by trying to modify the FSM we are specifying. We consider the modification of the source code out of the scope of this thesis. Second, VHDL already provides event driven simulation, and allows us to examine the affects of asynchronous events without modifying the Reacto simulator or Reacto specifications.

While either the intermediate state or event driven proposals could allow us to investigate asynchronous events, we assert that either modification we can undertake complicates R-Specs considerably. We resist the urge to improve Reacto's simulation capabilities by modifying the state machine at the expense of the specifications clarity and meaning. And, we leave intermediate states and more complicated event driven simulation out.

In comparison, we consider the Wait transition (described in Section 4.1.1) a necessary R-Spec complication since we cannot think of a better way to model response-response constraints correctly. We propose other enhancements to improve Reacto's capabilities without affecting the specifications clarity and meaning in Section 8.2.1.

There are undoubtedly more complicated systems than our example systems which may force us to make extensions to the Reacto model. As it is, our Reacto FSM as we have defined it is a "filing cabinet" allowing us to organize the theoretical process model as described in Levi and Agrawala (25:85). In our Mealy machine model, each transition represents a process. Computation time is transition delay. Begin constraint is the time the transition predicate becomes true. Process period is the highest frequency of events the transition is expected to respond to. Process deadline is defined by the start time plus the time constraint the transition is meeting. Now, we turn our attention to illustrating the Reacto extensions we have defined with our example problems.

4.2 Applying Reacto

We now demonstrate the Reacto portion of our methodology with the example problems. The complete Reacto source code for the cruise control and lift controller problems is contained in Volume II of this thesis[4]. We use sections of the code throughout this discussion, but we invite the reader to use Volume II for clarification.

We break applying Reacto down into three activities and the basic steps to accomplish each activity:

1. Build the abstract FSM.
   (a) Create FSM states.
   (b) Create FSM transitions.
2. Specify FSM behavior.
   (a) Define constraints and transition delays.
   (b) Define special data types.
   (c) Define input and output variables.
   (d) Write the *get-input* function.
   (e) Write state assertions.
   (f) Define transition behavior to satisfy State Assertions.
3. Verify and Validate FSM.
   (a) Compile and Verify the R-Spec, and make corrections.
   (b) Simulate the R-Spec, and make corrections.

We use Reacto's graphical editor to build the abstract FSM, then we use the graphical editor or text editor to fill in the underlying behavioral details of the FSM, and finally

[4]Volume II is available from the Air Force Institute of Technology, School of Engineering, at Wright-Patterson AFB, Ohio 45433-5000. Contact Major Kim Kanzaki at (513) 255-3708 to request a copy.

we examine and modify the FSM until it is an accurate formal specification of the behavior implied by the written English problem description. In the following paragraphs we describe each activities basic steps by applying each step to either the cruise control or lift controller problem.

4.2.1 Build the abstract FSM The first step in creating the R-Spec is to use the behavioral description from the informal specification to hierarchically define the states of the system from the most abstract to the most detailed. Figure 24 depicts our Reacto cruise control FSM states. The most abstract state is *Cruise-Control.* Within Cruise-

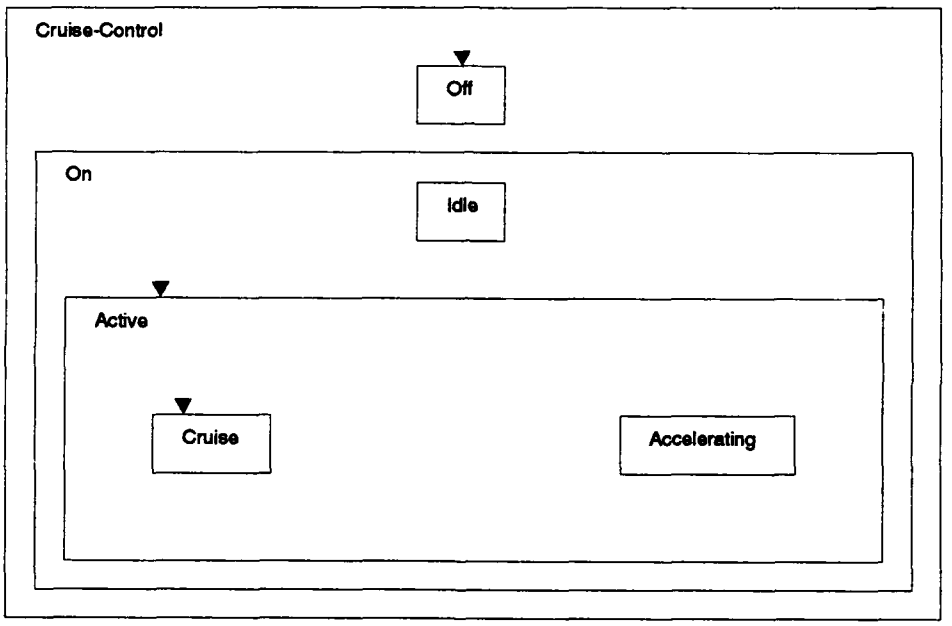

Figure 24. Reacto Cruise Control FSM States

Control the primitive default state *Off* and the superstate *On* states are the next most abstract. Within the On state, the system may be either *Active* or *Idle.* Finally at the lowest level of abstraction, the cruise control may be in the *Cruise* or *Accelerating* primitive states. The darkened-in ∇ symbol marks the default states.

The next step is to define a set of transitions on the states. The Transitions are the means to specify action in the Reacto model, and we assign them to accomplish the behavior described in the informal specification. Figure 25 depicts our Reacto cruise control FSM without its wait transition.

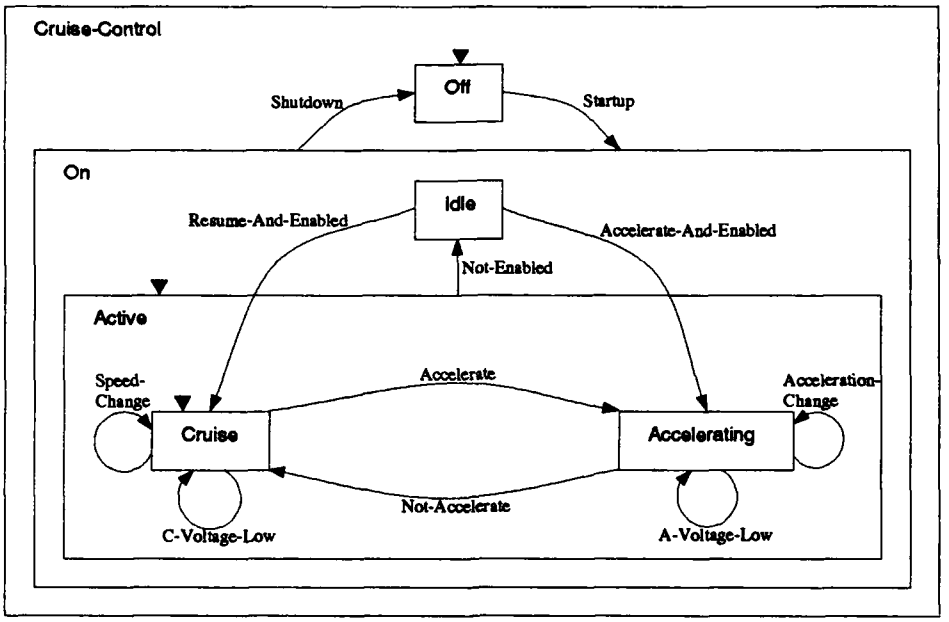

Figure 25. Reacto Cruise Control FSM

4.2.2 Specify FSM Behavior The first step to specify the details of FSM behavior is to define the system constraints and to associate transitions with those time constraints. When we specify transitions to accomplish time constrained behavior we must associate that behavior with the correct time constraint from the specification. Cruise control timing constraints are literally spelled out in Chapter III, Table 2, and are easily mapped to declarations in the cruise-control-auxiliary.re file as shown in Figure 26.

```
constant Start-Cruising-Time-Limit      : integer = 500
constant Start-Accelerating-Time-Limit  : integer = 500
constant Stop-Accelerating-Time-Limit   : integer = 500
constant Idle-Time-Limit                : integer = 500
constant Turn-Off-Time-Limit            : integer = 500
constant Speed-Change-Time-Limit        : integer = 1000
constant Acceleration-Change-Time-Limit : integer = 1000
constant Max-Volt-Change-Per-ms         : real    = 0.8/1000.0
```

Figure 26. Reacto Cruise Control Timing Constraints

Consider transition *Startup* in Figure 25; Startup turns on the cruise control under the conditions specified in the informal specification, and must meet the first time constraint from Chapter III, Table 2. Startup's delay is associated with that constraint by the following Reacto auxiliary file constant declaration:

```
constant Startup-Delay : integer = Start-Cruising-Time-Limit
```

Startup moves the Cruise Control from the Off state to the On state within 500 milliseconds (ms). Since On state's default state is Active, and Active's default state is Cruise, Startup moves the cruise control to the primitive state Cruise.

As shown with Startup, we initially set the delay of all transitions accomplishing time constrained work at the maximum allowed by the constraint. We use the maximum allowed by the constraint understanding that if maximum timing constraints are not violated under maximum delays, they are also satisfied by shorter delays[5]. If there are *dependencies* between transitions (two transitions interfere with each other such that they fail to meet time constraints) we may modify the delays such that time constraints are not violated thus pointing out transition dependencies. For transitions accomplishing work that is not time constrained, we set their delays to the simulation granularity, which for the cruise control is one millisecond. This allows us to detect dependencies between constrained and unconstrained transitions, modeling the fact that in the real world, all work takes time.

The next step is to define special[6] data types. Since the cruise control uses only integer, real and boolean data types, we define no special data types for it. In the lift problem however, we used special data types as shown in Figure 27. In Figure 27 types symbol, set, seq, and tuple are predefined Refine data types. Refine *symbols* correspond to natural language names (30:3-40), and they hold values like 'Up or 'Down. They are similar to VHDL's enumerated data types, except symbols can hold elements that are not predefined in some list. A Refine *set* is a finite, unordered, non-repeating collection of homogeneous elements (30:3-56). A Refine *seq* ("sequence"), is a totally ordered collection of homogeneous, possibly repeated elements (30:3-77). A Refine tuple is a finite, ordered collection

[5]Delays may not be less than any applicable minimum constraint.
[6]Ordinary data types are integer, real, or boolean.

```
type Lift-State-Type    = symbol
type Motor-Control-Type = symbol
type Direction-Type     = symbol
type Floor-Set-Type     = set(integer)
type Destination-Type   = seq(boolean)
type Summons-Type       = tuple(Floor     : integer,
                                Direction : Direction-Type)
type Summons-Set-Type   = set(Summons-Type)
type Status-Type        = tuple(State     : Lift-State-Type,
                                Floor     : integer,
                                Direction : Direction-Type)
type Status-List        = seq(Status-Type)
type Summons-List       = seq(Summons-Set-Type)
type Lift-Set-Type      = set(integer)
type Log-Type           = seq(integer)
```

Figure 27. Reacto Lift Special Data Types

of heterogeneous elements (30:3-107). Tuples are like records in typical programming languages. We define these special data types in a special source file, *lift-types.re* because the two R-Specs of the lift controller problem (lift and schedule-lifts) both use them. Putting the type declarations in a common file for both R-Specs insures they can communicate with each other using these types in VHDL.

The next step to applying Reacto is to declare global variables in the auxiliary file representing each FSM input and output. This is readily accomplished by referencing the FSM's SADT diagram. For example, Figure 28 shows the cruise control inputs and outputs corresponding to those illustrated earlier in Chapter III, Figure 13.

We do not use Reacto interface variables because interface variables make the mapping to VHDL harder. Using global variables allows us to implement transition predicates orthogonally in Reacto and VHDL. A mapping in VHDL for Reacto's empty-interface-variable?(x) predicate is difficult to conceive because VHDL signals always have a value. In Reacto, if we attempt to read an empty interface variable which has a defined producer function, the simulator automatically calls the producer function to get a value. In VHDL, signal values are updated under the control of the writer process and the system clock, not

```
%% Inputs
 var ACCELERATION        : real    = 0.0
 var ACTIVATE            : boolean = false
 var BRAKING             : boolean = false
 var CURRENT-SPEED       : real    = 0.0
 var DEACTIVATE          : boolean = false
 var ENGINE-RUNNING      : boolean = false
 var IN-TOP-GEAR         : boolean = false
 var RESUME              : boolean = false
 var STOP-ACCELERATION   : boolean = false
 var START-ACCELERATION  : boolean = false
%% Outputs
 var ACTUATOR-VOLTAGE    : real = 0.0
```

Figure 28. Reacto Cruise Control I/O

the reader process. In Reacto, input is controlled by the reader, hence we define our inputs as global variables and write a special get-input function to provide a clearer mapping to VHDL.

The next step is to create a get-input function in the auxiliary file. The get-input function works with the Reacto simulator. It drives the inputs, and reports the outputs in the Reacto-Emacs window during simulation. Figure 29 is part of the Reacto source code for the cruise control's get-input function augmented with line numbers for the following discussion. The format function called in line 4 outputs the current time and the value of the cruise control output *Actuator-Voltage*. The while loop (lines 6 through 16) repeatedly asks for an input variable name or simulation control command until the simulator inputs a null string or "go". The if statement (lines 8 through 16) determines what name or command is input each iteration, and performs the appropriate action. It calls input functions to get new input values, and other functions to control the simulation— e.g,. "mode" tells the simulator to toggle from the interactive to file simulation and visa versa. As the simulator changes input values, get-input echoes the current simulation time, and the new input value, producing a log of simulation events. Get-input allows us to update none, any, or all of the system inputs every time it is called. Our implementation of get-input also allows us to input from a text file and output to a text file. To allow input

```
1 function get-input() =
2 let (V-Name: string = "none")
3 % DISPLAY OUTPUT
4 format(t, "~D ms Voltage            = ~D.~%",Clock,Actuator-Voltage);
5 % UPDATE INPUT
6 (while V-Name ~= "" do
7  V-Name <- Read-String("Input name, mode, shut-off, ?  or <cr>");
8  (if V-Name = "acceleration" then
9       ACCELERATION <- Read-Real("New Acceleration?");
10      format(t, "~D ms acceleration       = ~D~%",Clock,acceleration)
11   elseif V-Name = "activate" then
12      ACTIVATE <- Read-Boolean("Activate?");
13      format(t, "~D ms activate           = ~A~%",Clock,activate)
     ...
15   else
16      format(t, "**~A**~%", V-Name))) % Echoes noninteractive inputs
```

Figure 29. Reacto Cruise Control Get-Input Function Source Code

events after every transition, we include a call to get-input as the last statement in every transition action.

The next step in applying Reacto is to write state assertions for each state describing what we want to be true about the system in each state. Figure 30 is the Reacto source code for the cruise control's Off state. In addition to the assertion concerning the Turn-

```
the-state Off
   assertion Actuator-Voltage = 0.0 &
             Clock - Turn-Off-Start-Time <= Turn-Off-Time-Limit
```

Figure 30. Reacto Cruise Control Off State Source Code

Off-Time-Limit time constraint, the Statement *Actuator-Voltage* = 0.0 corresponds to Requirement *R4* in the original cruise control informal specification from Section 3.3.1.2. The more requirements we express in these assertions, the more we verify about our R-Spec. Reacto's verifier and simulator detect R-Spec errors using these assertions. They point out when and where the R-Spec fails, enabling us to improve the R-Spec accordingly.

We recommend setting assertions for states with no assertions "true", to avoid problems with verification failing.

The next step is to fill in the details of transition behavior which makes the state assertions true. Figure 31 is the Reacto source code for the cruise control's Startup transition augmented with line numbers to aid in our discussion. We write transitions conforming to

```
1 "Startup"
2 a-transition off-transition-1 from Off to On
3  predicate Activate & Current-Speed >= the-real(30.0)
4              & In-Top-Gear & ~Braking & Engine-Running & ~Deactivate
5  action Start-Cruising-Start-Time <- Clock;
6         Turn-Off-Start-Time <- Start-Time-Reset;
7         Clock <- Clock + Startup-Delay;
8         Old-Speed <- Current-Speed;
9         Target-Speed <- Current-Speed;
10        Cruise-Voltage <- Get-Cruise-Voltage(Target-Speed,
11                                             Current-Speed);
12        Actuator-Voltage <-
13           Cruise-Voltage
14             Min (Max-Volt-Change-Per-ms * Startup-Delay);
15        Get-Input()
16  priority 20
```

Figure 31. Reacto Cruise Control Startup Transition Source Code

the following template (line numbers are shown in Figure 31):

1. Transition Label (line 1).
2. Transition Name and from, to states(line 2).
3. Predicate (lines 3-4).
4. Action (lines 5-15)
   - Set and reset start time logs (lines 5-6).
   - Update clock (line 7).
   - Transition "work" (lines 8-14).
   - Call get-input (line 15).
5. Transition Priority (line 16).

We must fill in the transition predicate (lines 3 and 4) and the actual transition behavior (lines 8 through 14) to meet the behavioral requirements of the informal specification. Startup starts the cruise control up when the user presses the Activate button and the other conditions defined by the predicate are true. To start the cruise control, Startup calls the function Get-Cruise-Voltage (line 10) which we define in the auxiliary file. Get-Cruise-Voltage returns the voltage that the cruise control should be at to correct any difference between the Target-Speed and Current-Speed. Since the cruise control is constrained not to increase voltage more than 0.8 volts per second, Startup uses the predefined lisp operator *min* (line 14) to set the output voltage to either the calculated voltage, or the voltage increase allowed by Startup-Delay. Min has two arguments, *Cruise-Voltage* on line 13, and the arithmetic expression on line 14.

When we specify behavior it is important to consider whether or not to use Refine's higher level abstractions like transforms, maps, sets and quantification on those sets in the R-Spec. Although these abstractions are quite powerful, there is a cost associated with the transformation to VHDL since VHDL is a procedural language without these abstractions. In the simpler cruise control problem, we use only procedural Refine statements, and the transformation process is straight forward. In the lift controller problem however, we use sets and quantification forcing us to define set types and operations on those types in VHDL making the transformation process more complicated. For this reason, we encourage the use of procedural Reacto statements where practical.

Finally, we establish the transition priority (line 16) based on what the informal specification requirements say about the priority of the work we are accomplishing. We recommend changing each transition's priority[7] to a unique number greater than zero to avoid inconsistency problems between the VHDL and Reacto simulation behavior. Reacto executes transitions of equal priority based on their order in the Refine knowledge base when their predicates are simultaneously true. Since we do not necessarily know what knowledge-base order they are in during our manual translation process, we may unintentionally define the order differently in VHDL. Reacto creates lines 1 and 2 when we save the R-Spec file after we create the graphical FSM.

[7]Default priority is zero.

4.2.3 Verify and Validate FSM The first step of the Verify and Validate FSM behavior activity is to compile and verify the R-Spec, and make any necessary corrections in transitions and state assertions. We refer the reader to the *Reacto User's Manual* (23) and *Reacto-Verifier User's Guide* (22) for specific instructions on compiling and using the Reacto-Verifier. An in-depth discussion of the Reacto-Verifier is beyond the scope of this thesis.

The final step in applying Reacto is to Simulate the R-Spec, examine its behavior, and make any necessary corrections in R-Spec transitions, state assertions and auxiliary file functions. During this step, we "Copy Assertions in the Focussed Spec to Runtime Checks[8]" prior to running the simulation, so that in addition to verifying the assertions with the Reacto-Verifier, we verify them during simulation. During initial simulation, we manually drive the R-Spec with simple tests like those described for the Relay FSM in Section 4.1.2, and we examine the basic behavior of the FSM. This exposes gross behavioral errors, but is quite tedious for more complicated test cases. After we are satisfied with the gross behavior, we create a simulation input file and use it to drive the simulation. This is valuable, because comparing the results of subsequent simulations under the same input conditions exposes errors caused by our "fixes" to other problems. Section 4.3 describes our test cases. The full set of test cases are in Appendices A and B. In Section 4.4 we describe some of the improvements we make as a result of applying Reacto and observing test case behavior.

4.3 Reacto Test Cases

Writing, simulating, and evaluating test cases is the primary means we have to verify and validate R-Spec behavior. In the following Sections, we describe the test cases we use to verify and validate the cruise control and lift-controller R-Specs. Our final R-Specs pass all of these tests, outputting the correct results, without failing any assertions. Since it is a good representative test case, we include a detailed test case, simulation input, and simulation output here for the *Activation Allowed Test*, and we refer the reader to

[8] A Reacto Edit Option.

Appendix A for cruise control test details and to Appendix B for details of lift test cases. We analyze the *Activation Allowed Test* output, and assert that our R-Specs satisfy similar analysis for all of our test cases. We refer the reader to Volume II of this thesis[9] for actual Reacto inputs and results of each test case.

4.3.1 Cruise Control Test Cases

Initialization Test Initially, the cruise control shall be in an off state. Therefore, upon startup, throttle actuator voltage shall be zero until speed is greater than 30 miles per hour and the driver command activate is asserted. Requirement implied.

Activation Denied Test There are three phases to this test. The first phase tests that the system shall not activate when the car is in top gear, the engine is running, but the speed is not 30 miles per hour (mph) or more. The second phase tests that the system shall not activate when the car's speed is 30 mph or more, the transmission is in top gear, but the engine isn't running. The final phase tests that the system shall not activate when the cars speed is 30 mph or more, the engine is running, but the transmission isn't in top gear. Requirements Tested: R1, R2, R3.

Activation Allowed Test This tests that the cruise control shall activate when the engine is running, the transmission is in top gear, and the speed is at least 30 mph. Requirements Tested: R1, R2, R3, C4. Test steps:

1. Set In-Top-Gear, Engine-Running to true
2. Set Braking to false
3. Set Speed to 30 mph
4. Toggle Activate between true and false
5. Set Speed to 29 mph
6. Is Actuator-Voltage greater than 0.0 volts?

Actual test input is shown in Figure 32. Get-input echoes lines containing "*****"

[9]Volume II is available from the Air Force Institute of Technology, School of Engineering, at Wright-Patterson AFB, Ohio 45433-5000. Contact Major Kim Kanzaki at (513) 255-3708 to request a copy.

```
Start Activation Allowed Test ****
in-top-gear
t
current-speed
30.00
engine-running
t
activate
t
go
activate
f
go
go
go
go
go
current-speed
29.00
go
go
go
shut-off
Stop Activation Allowed Test ****
```

Figure 32. Reacto Cruise Control Activation Allowed Input

to the simulation output file. Lines containing "go" direct the get-input function to cease inputting events and return, allowing the FSM to complete the current transition, evaluate state assertions, and evaluate transition predicates for the next execution. Lines containing input variables are followed by the values to assign to those inputs. For example, the line "current-speed" is followed by the line "30.00", indicating that current-speed should be set to 30 mph. The lines "activate" and "t" cause the activate input to be set to true after engine-running is set to true. When get-input reads the line "shut-off", it calls the simulation control function Shut-Off to turn the cruise control off in preparation for the next test.

Actual *Activation Allowed Test* output is shown in Figure 33. The output lines containing "Go..." are written by the get-input function each time the simulation file indicates it is finished inputting events for the current time. Note that get-input logs all input and output events along with the time of each event. The output "Shut-Off Complete" comes from the simulation control function Shut-Off which turns off the cruise control in preparation for the next test. The reason there is no event for the *Set Braking to false* step, is that it is accomplished by the Shut-Off function during the previous test.

We must analyze this output to determine if the cruise control functions correctly. At 6000 ms, we tell the cruise control to activate. Since the transmission is in top gear, the speed is at least 30 miles per hour, the engine is running, and the brakes are off, it turns on the cruise control. It turns the cruise control on 250 ms later at 6250 ms, when the output voltage is set to 0.2 volts. It does not increase the voltage to 4.0 volts[10] immediately because this would violate the maximum allowed voltage increase of 0.8 volts/second. Instead, every second it increases the voltage by the amount allowed until it reaches the 4.0 volts. At 11250 ms, we change the current speed to 29 miles per hour, and the cruise control responds by raising the voltage to increase the speed. Once again it does it subject to the 0.8 volts/second constraint, updating to the required 6.0 volts by 14250 ms. At 14250 ms, we shut the cruise control off in preparation for the next test. These are the expected results,

[10]The engineer's algorithm says that 4.0 volts are required to maintain the current speed.

```
**Start Activation Allowed Test ******
6000 ms in-top-gear          = T
6000 ms current-speed        = 30.0
6000 ms engine-running       = T
6000 ms activate             = T
Go...
6250 ms Voltage              = 0.2.
6250 ms activate             = NIL
Go...
7250 ms Voltage              = 1.0.
Go...
8250 ms Voltage              = 1.8.
Go...
9250 ms Voltage              = 2.6.
Go...
10250 ms Voltage              = 3.3999999.
Go...
11250 ms Voltage              = 4.0.
11250 ms current-speed        = 29.0
Go...
12250 ms Voltage              = 4.8.
Go...
13250 ms Voltage              = 5.6000004.
Go...
14250 ms Voltage              = 6.0.
Shut-Off Complete
14750 ms Voltage              = 0.0.
**Stop Activation Allowed Test ******
```

Figure 33. Reacto Cruise Control Activation Allowed Output

and no Reacto assertions are violated, so this version of the cruise control passes the *Activation Allowed Test.*

Deactivation Test This tests that the cruise control shall shut off when the driver presses deactivate. Requirements tested: R1, R4.

Acceleration Test This tests that the cruise control shall accelerate the car when the driver presses Start-Accelerating, and that it stops accelerating when the driver presses Stop-Accelerating. It also tests that the acceleration shall be approximately $1 mph/sec$. Requirements Tested: R5, C1, C4, C5, C6, C7, C8.

Resume Test This tests that the cruise control shall resume the previous speed when the driver presses Resume after Braking or not In-Top-Gear. Requirements Tested: R6, R8, R10, R11, R12, C4.

Downhill Test This tests that the cruise control shall attempt to maintain the selected speed by decreasing actuator voltage to minimum when the current speed remains more than 2 mph above the selected speed. Requirements Tested: C2, C4.

Uphill Test This tests that the cruise control shall attempt to maintain the selected speed by increasing actuator voltage to maximum when the current speed remains more than 2 mph below the selected speed. Requirements Tested: C3, C4.

Other Tests The Reacto test cases Breaking During Activate Delay Test, Breaking During Activate Asserted Test, Breaking After Activate B4 Cruise Test, Resume during Breaking Test, and Deactivate overlaps Resume Test are included in the Reacto test cases, but they are asynchronous event tests, meant for use with VHDL. Figure 34 is the output from the Reacto Breaking During Activate Delay Test, and as you can see, we input the Braking event after the cruise control activated at 104750 ms, because we cannot input an asynchronous event between 104500 ms and 104750 ms.

4.3.2 Lift Controller Test Cases Since the lift controller is a more complicated system[11], we do not show the details of an example test case like the cruise control. Instead,

[11]The output of the first lift test *All Summons* is 197 lines long.

```
**Start Breaking During Activate Delay Test **
104500 ms in-top-gear          = T
104500 ms current-speed        = 45.0
104500 ms engine-running       = T
104500 ms activate             = T
Go...
104750 ms Voltage              = 0.2.
104750 ms braking              = T
Go...
105000 ms Voltage              = 0.0.
105000 ms activate             = NIL
Go...
106000 ms Voltage              = 0.0.
106000 ms braking              = NIL
Go...
107000 ms Voltage              = 0.0.
Shut-Off Complete
107500 ms Voltage              = 0.0.
**Stop Breaking During Activate Delay Test **
```

Figure 34. Reacto Cruise Control Breaking During Activate Delay Test Output

we generally describe each test, referring you to Appendix B, where you can see every detail of each test. In the following Section, we describe the lift test cases, and in Section 4.3.2.2, we describe the schedule lifts test cases.

4.3.2.1 Lift Tests The following cases test the behavior of a single lift in a four-floor building.

All Summons Test tests lift's ability to handle all Summons from every floor. In the real world, this corresponds to people simultaneously pushing every summons button on every floor requesting the elevator pick them up.

All Destinations Test tests lift's ability to handle people inside the lift simultaneously pushing all destination buttons.

Emergency Button Test tests lift's response to emergency button events.

Mixed Destinations and Summons Test is a real-world scenario that tests lift's ability to handle both summons requests and destination requests together.

Timeout Test tests lift against the response-response constraint which specifies the amount of time the lift waits for a destination in the current direction if there are no destinations already in that direction.

4.3.2.2 Schedule Lifts Tests The following cases test the behavior of schedule lifts when configured as a four-floor, four-lift system.

Off State Test tests schedule lifts behavior when all lifts are off.

All Summons 1 Lift Test tests schedule lifts behavior when only one lift is available to handle summons requests from all floors in all directions.

Idle Schedule Test tests schedule lifts behavior when all elevators are available, and only one is idle.

All Summons Test tests schedule lifts behavior when all lifts are available to handle summons requests from all floors in all directions.

As with the cruise control, we analyze all test output, and assert that our final lift and schedule-lifts R-Specs satisfy thorough analysis for all of our test cases. During the iterative validation and verification process however, the R-Specs did not always perform as expected, resulting in many specification improvements. We discuss examples of these improvements in the next section.

4.4 Reacto Specification Improvements

Having described our Reacto augmentations, the application of Reacto to our example problems, and the test cases we use to verify and validate the FSM, we turn our attention to assessing the resulting specification improvements.

At the highest level of abstraction, the specifications for both the cruise and lift control systems are improved by simply specifying the FSM. We demonstrate this by three examples, one for each FSM. Examining Figure 35 allows us to point out an improvement made over the informal cruise control specification. The Accelerate-And-Enabled

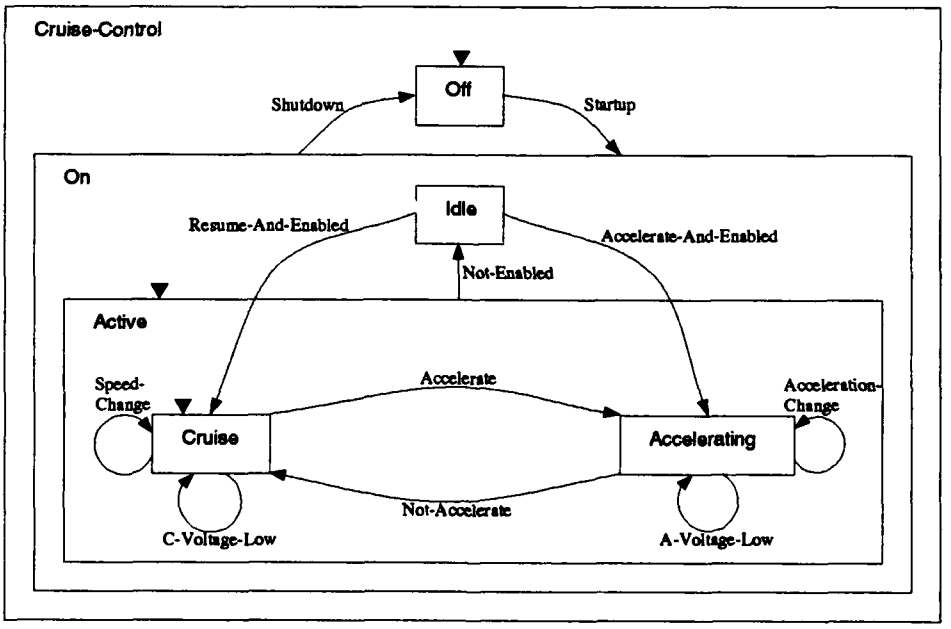

Figure 35. Reacto Cruise Control FSM

transition from Idle to Accelerating clarifies an ambiguity in the informal specification. Requirement *R5* in the informal specification states that "Start Accelerating causes the

system to accelerate the car at a comfortable rate until Stop Accelerating occurs, when the system holds the car at this new speed," but it does not say if Start Accelerating has this effect only when the car is currently cruising or if it applies when the cruise control had been cruising but the driver momentarily hit the brakes or the transmission went out of high gear (i.e., shall the cruise control allow acceleration from the Idle state?). Our R-Spec FSM clarifies requirement *R5* allowing Acceleration from the Cruise state via the Accelerate transition, and from the Idle state via the Accelerate-And-Enabled transition.

Specifying the lift control system FSMs for schedule lifts and lift also clarifies ambiguities in the lift controller informal specification. Figure 36 is the lift FSM. Eickmeier

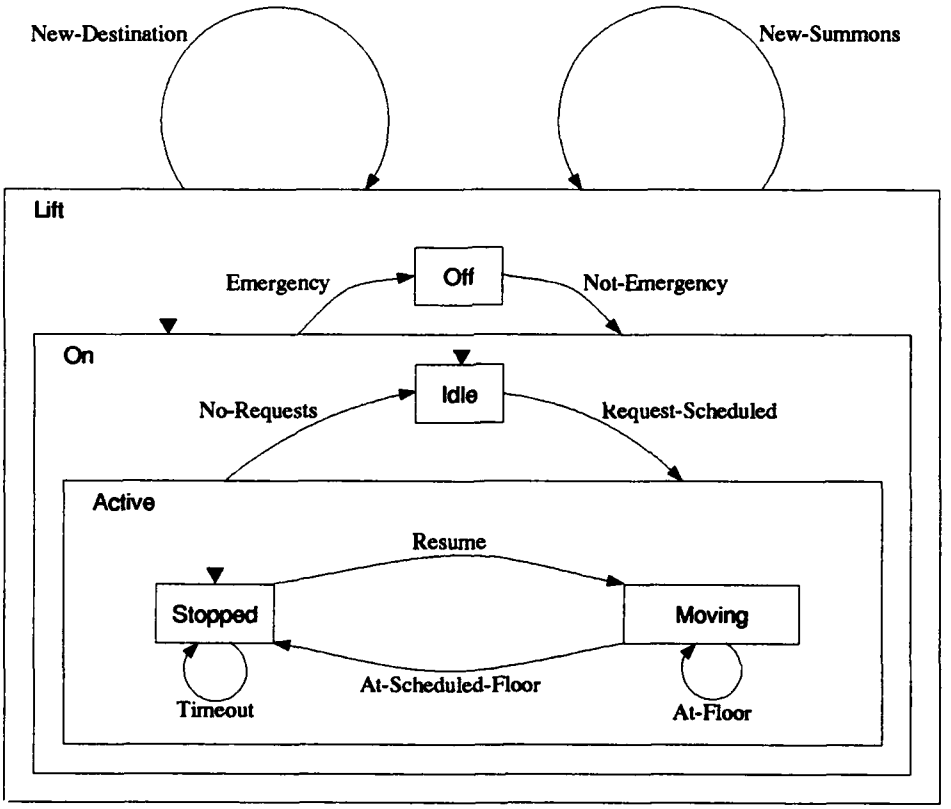

Figure 36. Reacto Lift FSM

points out that the informal lift controller specification is ambiguous with respect to lift behavior when the emergency button is pressed (12:92). The informal specification does not specify if the lift responds to destination buttons pressed while the emergency button is pressed. In Figure 36 the New-Destination transition is active during all states, including

Off, therefore, we specify that the lift schedules destinations when destination buttons are pressed regardless of the position of the emergency button. If we wish to specify otherwise, we simply move the New-Destination transition inside the Lift state, making it active only in the state On and its substates.

Similarly, the schedule lifts FSM as depicted in Figure 37 specifies that Summons buttons are not responded to unless there is at least one lift available[12] because the Summons-Button-Event transition is only active from the On state.

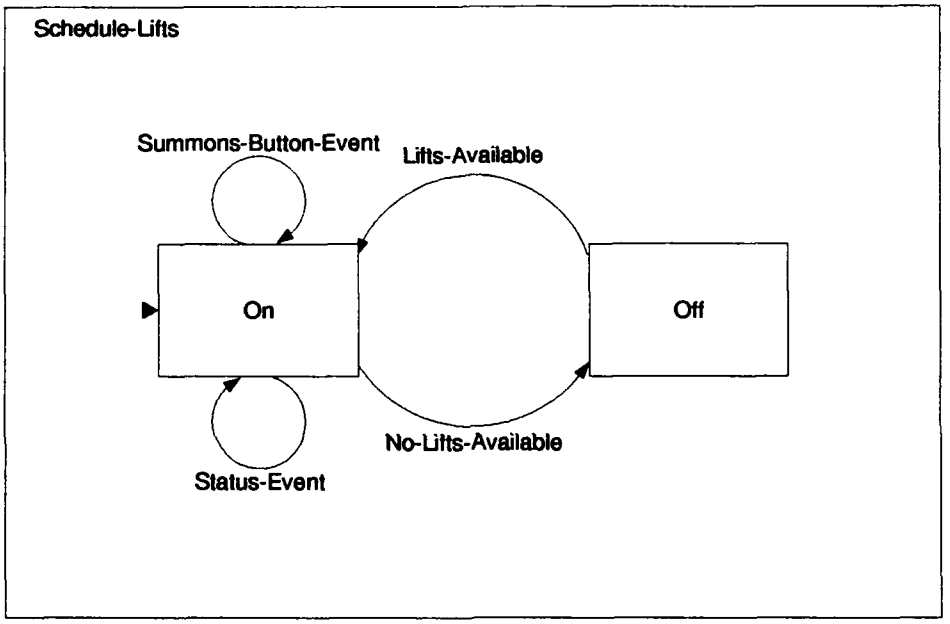

Figure 37. Reacto Schedule Lifts FSM

In addition to specification improvements made by creating the FSMs themselves, our R-Specs clarify more ambiguities. Eickmeier pointed out two additional ambiguities in the lift controller problem (12:92).

- Does a lift stop immediately when the emergency button is pressed, or does it travel to the next floor before stopping?

[12] A lift is available if its emergency button is not pressed.

- Is the door open when a lift has no summons or destination requests (i.e., the lift is idle)? Or worded another way, must a Summons request be issued to open the door of an idle lift?

We resolve Eickmeier's first ambiguity in the lift spec's predicate as shown in Figure 38. When the lift is at a floor, floor sensor is greater than zero, otherwise, the lift

```
"Emergency"
   a-transition ON-TRANSITION-2 from ON to OFF
   predicate Emergency-Button and Floor-Sensor > 0
```

Figure 38. Reacto Lift Emergency Transition Predicate

is between floors, hence we specify that the lift stops at the next available floor and not between floors.

Although we do not explicitly model door opening and closing in our lift controller as explained in Section 3.3.2.1, we clarify Eickmeier's second ambiguity using states Off, Idle, and Stopped as shown in Figure 36. States Off and Stopped are states where the lift doors are open. In an emergency, the lift is in state Off, and the doors remain open. Doors are closed in the idle state, and when a request is scheduled, the lift transitions to the Active state via the transition Request-Scheduled. Active's default state is Stopped. Hence we specify that lift doors are closed when a lift has no Summons or destination requests, and that they open upon receiving a Summons request.

Any remaining ambiguities are resolved according to the behavior specified for the transitions and the functions we define in the auxiliary files. For example, Figure 39 shows the mathematical algorithm used to calculate the voltage that the cruise control outputs while in state cruise in accordance with the mechanical engineer's recommendations from (17:291). This algorithm is not included in the informal requirements. Some may consider it a step toward implementation, but it is necessary to provide an executable model of the requirements. Reacto provides considerable flexibility, allowing one to specify behavior more abstractly or in greater detail. A good example of this flexibility is our scheduling algorithm Pick-A-Lift from the schedule-lifts auxiliary file. Our Pick-A-Lift

```
function Get-Cruise-Voltage (Target : real, Actual : real) : real =
   let(delta-speed : real = target - actual,
       voltage     : real = 0.0)
   (if delta-speed > 2.0 then
      voltage <- 8.0
   elseif delta-speed >= -2.0 & delta-speed <= 2.0 then
      voltage <- 2.0 * (delta-speed + 2.0)
   else voltage <- 0.0);
   (voltage)
```

Figure 39. Reacto Get-Cruise-Voltage Function

function (see *lifts-auxiliary.re* file in Volume II) is fairly detailed, taking approximately 72 lines of Refine statements without comments and declarations. If we wish to specify the scheduling function more abstractly, we could used the Reacto arb (extracts an arbitrary element from a set) function on the set of available lifts, leaving virtually all the details up to the implementation. If we want to be more sophisticated we can enhance Pick-A-Lift accordingly.

In the previous paragraphs, we describe some of the informal requirements ambiguities we resolve by specifying the FSMs and completing the details of the R-Specs. Now we describe an example behavioral specification improvement made to the interface between two FSMs a result of going through the process of creating and simulating the R-Specs. Originally, our Lift and Schedule-Lifts interface included a *Status-Type.Summons* as depicted in Figure 40. Originally, the Lift FSM used Status-Type.Summons to tell Schedule-Lifts that the Lift received the Summons and added it to it's Summons schedule. Currently, we eliminate Status-Type.Summons from the interface, and centralize Summons scheduling in Schedule-Lifts[13]. We made this change because the Reacto simulations and modifications to Schedule-Lifts were very complex with the Old interface. We had trouble deciding whether the Lift should delete Summons from the Lift schedule or if Schedule-Lifts should. After moving the function back and forth between the R-Specs twice, we realized

[13]Destination scheduling is still done in Lift, since every Lift must take the people inside it to their destination.

```
type Status-Type = tuple(State     : Lift-State-Type,
                         Floor     : integer,
                         Direction : Direction-Type,
                         Summons   : Summons-Set-Type)
```

Old Type Definition

```
type Status-Type = tuple(State     : Lift-State-Type,
                         Floor     : integer,
                         Direction : Direction-Type)
```

Current Type Definition

Figure 40. Reacto Status-Type Definitions

that we had complicated the specification to the point that Lift and Schedule-Lifts were too closely coupled. Reacto's visual simulation environment and its provision to rapidly make these changes made it easy to discover and resolve this coupling problem within a single day. The end result is a clearly defined interface and division of responsibility between Lift and Schedule-Lifts.

This completes our discussion about example Reacto specification improvements for our two problems, we turn now to the next activity in our methodology, transforming from Reacto to VHDL.

4.5 Conclusion

In this chapter we describe augmenting Reacto with time and applying Reacto to our example cruise control and lift controller problems.

Our Reacto augmentations include:

- Clock
- Time Limit, Min Time and Duration Constraints
- Transition Delays

- Start Time Logs and Timers
- Wait Transition

They make it possible to express time constraints and simulate the behavior of real time systems in Reacto as described in our simple Relay example. We discuss that we could model asynchronous events in Reacto if we add intermediate states to our FSMs or implement an event driven simulation, but we choose to use VHDL's event driven simulator to investigate asynchronous events instead.

We apply Reacto to the Example problems, walking through the following steps with our example problems:

1. Build the abstract FSM.
   (a) Create FSM states.
   (b) Create FSM transitions.
2. Specify FSM behavior.
   (a) Define constraints and transition delay.
   (b) Define special data types.
   (c) Define input and output variables.
   (d) Write the *get-input* function.
   (e) Write state assertions.
   (f) Define transition behavior to satisfy State Assertions.
3. Verify and Validate FSM.
   (a) Compile and Verify the R-Spec, and make corrections.
   (b) Simulate the R-Spec, and make corrections.

We describe the test cases that we use to validate and verify the cruise control and lift controller specifications in Reacto simulations. Finally, we make significant behavioral improvements in both the cruise control and lift controller specifications, updating the R-Specs until they produce the correct results without assertion errors while simulating our test cases. We are able to make these improvements quickly and easily because of Reacto's powerful abstractions and visual simulation environment.

V. Reacto to VHDL Transformation

In Chapter III we identified the Reacto to VHDL transformation as a significant supporting activity, and indeed it is a significant portion of our research. The process of defining the transformation lends insight into the differences between the languages, and emphasizes the difficulties of describing the relationships of system inputs and outputs with respect to time.

We attempt to provide a general behavior preserving mapping from Reacto FSMs to VHDL FSMs, and we consider the transformation of the cruise control, lift, and schedule lifts FSMs examples of this mapping. A more rigorous and formal mapping between the two is necessary to insure that all Reacto FSMs can be transformed into VHDL in a behavior-preserving manner. We consider that rigorous mapping an essential step before automating the transformation from Reacto to VHDL, but we do not accomplish that mapping because we consider it beyond the scope of this thesis. It is worthwhile to note that once the mapping is defined for Reacto to VHDL, the basis will exist to examine mappings from Reacto to any state-based language such as Ada, C, Pascal, etc...

We show the general mapping of Reacto elements to VHDL elements in Table 4. We use the notation X $\longrightarrow$ Y in the table to indicate the mapping of a Reacto element to an element of VHDL.

The following Reacto elements are used to control the generation of the VHDL FSM controlling case statement and transition procedures, but they are not mapped to a specific object in VHDL:

- State Attributes
  - substates
  - initial-state
- Transition Attributes
  - from-state
  - to-state

Table 4. Reacto to VHDL Mapping

Reacto	⟶	VHDL
States		
Name	⟶	Current-State declaration
Own-Vars	⟶	Signals and Variables
Assertion	⟶	Assertion Procedure
Runtime-Check	⟶	Assertion Procedure
Transition		
Predicate	⟶	if-then-else predicate
Label	⟶	Procedure Name
Action	⟶	Transition Procedure Body
Type Declarations	⟶	Type Declarations
Sequences	⟶	Arrays
Sets	⟶	Integer Sets
Tuples	⟶	Records
Input Variables	⟶	Entity declaration, **in** Port
Output Variables	⟶	Entity declaration, **inout** Port and Variable Declaration
Global Variables	⟶	Signal and Variable Declarations
Constants	⟶	Constants
Functions	⟶	Functions
Quantification	⟶	Functions

- history-flag
- priority

We do not map the Reacto call-state mechanism because its function is to implement textual substitution (23:6), which is something we do not use in our sample problems. Additionally, we do not map Reacto Interface variables as explained in Section 4.2.2.

In the following paragraphs, we use examples to illustrate some differences between Reacto and VHDL and to discuss the problems of describing temporal behavior. One of the differences is so significant, we address it separately in Section 5.1. We organize our examples by associating them with the process of generating each VHDL source code section in the following order:

- Generating Declarations
- Generating Setup and Cleanup Procedures
- Generating VHDL Functions
- Generating Assertion Procedures
- Generating Transition Procedures
- Generating the FSM Process Body

The actual VHDL source code for the cruise control and lift controller problems is contained in Volume II, and we invite you to reference it for clarification as necessary.

After we discuss generating each section, we evaluate the benefits and complications of automating the transformation in Section 5.8.

5.1 Preemptive Execution Model

We must explain one fundamental difference between the R-Spec and V-Spec models before we can describe the translation process effectively. In Reacto, we control the clock with the FSM transitions. In VHDL, the simulator controls the clock. Therefore, events in the VHDL model may occur independently of transition execution, i.e., we can model asynchronous events in VHDL. Events are no longer constrained to happen at discrete

times dictated by transition updates of the clock. This allows us to model what we call a *Preemptive Execution Model* (PEM) in VHDL. In the PEM, a transition executing at time T changes the internal and external state of the FSM at time $T+$ *Transition-Delay.* If subsequent input events occur before time $T+$ *Transition-Delay* such that a higher priority transition predicate becomes true, the higher priority transition executes preempting the scheduled state changes of the lower priority transition. Investigating the effects of asynchronous events in the VHDL PEM sheds a great deal of light on transition dependencies, and we discuss those revelations in Section 6.5.2.

5.2 Generating Declarations

The first required VHDL declaration is the entity declaration. It declares the FSM entity and describes its interface to the outside world. We use the most abstract state from the R-Spec to name the entity, and we use inputs and outputs declared in the auxiliary file to generate the entity declaration. Figure 41 shows the cruise control inputs and outputs corresponding to those generated from Figure 28. VHDL requires that port direction be

```
entity Cruise_Control is
   port ( In_Top_Gear,
          Braking,
          Engine_Running,
          Deactivate,
          Start_Acceleration,
          Stop_Acceleration,
          Activate,
          Resume             : in boolean := false;
          Current_Speed      : in real    := 0.0;
          Acceleration       : in real    := 0.0;
          Actuator_Voltage   : inout real := 0.0 );
end Cruise_Control;
```

Figure 41. VHDL Cruise Control Entity Declaration

specified. We specify the *in* direction for all FSM inputs, and *inout* for all FSM outputs since we read them to make state assertions and to use their current value in calculations.

Note that Reacto identifiers contain hyphens, but not underscores and VHDL identifiers contain underscores but not hyphens. We change all hyphens to underscores as we translate. Additionally, we cannot use any Reacto or VHDL reserved words as identifiers. For example, we change Schedule-Lifts' state name from "On" in the R-Spec to "On1" in the V-Spec because the word "on" is a reserved word in VHDL.

Although not illustrated in our cruise control example, connecting different entities in VHDL with special data type signals requires declaration of these types in a package. In our VHDL package *Lift_Types* we define the special data types necessary to connect lift and schedule lifts together. Any entity that uses special data types requires a separate package defining the special data types because at a minimum, we must connect the FSM entity to a Testbench entity for simulation. We compile VHDL type declaration packages and "use" them via statements like *use work.Lift_Types.all* in every source file where those types are referenced. A separate package is not necessary in Reacto because we do not connect Reacto state machines together, but we prepare for the VHDL transformation by generating and compiling a *lift-types.re* file which includes the necessary declarations.

Once we generate the entity declaration, we begin generating the architecture which describes the behavior of the entity declaration. The first part of the architecture is its declarations section. We generate VHDL constant and signal declarations from the Reacto auxiliary file constant and variable declarations, and also from Reacto spec file state-owned variable declarations. Example declarations are illustrated in Figure 42. Note that in Figure 42 we change the type of time objects from integer to the predefined VHDL type *time*. We also change the set(integer) declaration to set(1 to Top_Floor) matching our VHDL implementation of integer sets.

Reacto auxiliary file declarations are global, but R-Spec state-owned variables are scoped lexically (23:4). This means that state-owned variables declared in state S1 are visible in S1 and substates of S1. In each VHDL FSM, all identifiers except dynamic function variables are global, requiring a mapping to unique VHDL identifiers for repeated Reacto state-owned variable identifiers. We repeat no identifiers in our example R-Specs, but we propose that creating unique identifiers in VHDL by prepending the state name

```
type Floor-Set-Type = set(integer)
```

Reacto Lift-Type.Re Declaration

```
constant Light-On-Time-Limit : integer = 100
var Timeout-Timer : integer = Max-Time-Reset
```

Reacto Auxiliary File Declarations

```
the-state LIFT
  own-vars( Dest-Schedule : Floor-Set-Type   = {})
```

Reacto Spec File Declaration

```
subtype Floor_Set_Type is set(1 to Top_Floor);
```

VHDL Lift_Types Declaration

```
constant Light_On_Time_Limit : time := 100 ms;
signal Timeout_Timer : time := Max_Time_Reset;
signal Dest_Schedule : Floor_Set_Type := (others => '0');
```

VHDL Architecture Declarations

Figure 42. Declarations Example

solves this problem— e.g, if Reacto state S1 owns variable A and Reacto state S2 owns variable A, they map to VHDL identifiers S1A and S2A.

We map Reacto declarations of type symbol to VHDL enumerated type declarations in the VHDL types package and corresponding VHDL object declarations of that type as shown in Figure 43. Every Reacto symbol object is compatible with every other Reacto

```
type Motor-Control-Type = symbol
var Motor-Control : Motor-Control-Type = 'Off
```

Reacto Symbol Type and Object Declarations

```
type Motor_Control_Type is (Up, Down, Off);
Motor_Control : inout Motor_Control_Type := Off;
```

VHDL Symbol Type and Object Declarations

Figure 43. Symbol Type Declaration Example

symbol object, and objects of type symbol can take on any symbol value. Therefore, generating VHDL type declarations for Reacto variables of type symbol requires searching the R-Spec for all of the values assigned to the variable. Similarly, we generate the VHDL state type declaration and the state signal declaration using the R-Spec primitive states in the Reacto machine as shown in Figure 44. Note that we initialize *Current_State* to the

```
type Lift_State_Type is (Off, Idle, Stopped, Moving);
signal Current_State : Lift_State_Type := Idle;
```

Figure 44. VHDL Lift State Declarations

R-Spec's default state *Idle*.

Initial values for Reacto's state-owned variables can be set to the value of R-Spec input variables. In VHDL however, inputs are signals and their values depend upon configurations. Since configuration values are not available at V-Spec compile time, VHDL does not allow initialization with the signal name. We make extensive use of the VHDL

"others" clause and our knowledge of what the R-Spec input initializations are to correctly initialize VHDL signals corresponding to R-Spec state-owned variables.

After we map all R-Spec constants and variables to V-spec constant and signal declarations, we are finished generating architecture declarations, and we proceed to generating the architecture body and its associated declarations.

The first declaration in the architecture body is the concurrent process that models the behavior of the FSM entity. The lift process declaration shown in Figure 45 is an example process declaration. We include all inputs in the process sensitivity list so that the

```
Lift : process
   (Dest_Buttons, Emergency_Button, Floor_Sensor,
    Outstanding_Requests, Summons,
    Current_State'transaction, Timeout_Timer)
```

Figure 45. VHDL Lift Process Declaration

process executes on every input event. We add *Current_State'transaction* to the sensitivity list so that the FSM examines the predicates of transitions in a new state immediately after it enters the new state. If we do not, the FSM stalls until another input event occurs even though more transition predicates may be true in the new state. We add all timers (like *Timeout_Timer* in Figure 45) to the sensitivity list making the FSM respond in accordance with the response-response constraints that generated the timer declarations.

In order to implement the PEM in VHDL, we map each R-Spec variable and output variable into a VHDL signal and a VHDL variable as shown in Figure 46. As described in the previous paragraphs, we declare all signals in the architecture declaration section. Now, we declare the variable in the process declaration section of the architecture body. These two declarations make it possible to keep internal and external state information consistent when transitions are preempted. Transitions execute in three steps. First, the transition copies the current value of the signal *Floor_Set* into variable *T_Floor_Set* by calling procedure Setup. Second, the transition updates T_Floor_Set executing the V-Spec statements transformed from the R-Spec transition action. Third, the transition calls

```
Floor-Set : Floor-Set-Type   = {}
```

Reacto Variable Declaration

```
signal   Floor_Set : Floor_Set_Type := (others => '0');
variable T_Floor_Set : Floor_Set_Type := (others => '0');
```

Corresponding VHDL Signal and Variable Declarations

Figure 46. Variable Transformation Example

special procedure Cleanup to schedule an update of signal Floor_Set with the current value of T_Floor_Set. The signal Floor_Set is updated by the VHDL simulator after transition delay if no subsequent updates are scheduled for Floor_Set by another higher priority transition.

5.3 Generating Setup and Cleanup Procedures

We generate Setup and Cleanup procedures to implement the PEM. The first statement in every transition is a Setup call, and the last statement is a Cleanup call. We use R-Spec output and local variables to generate both the Cleanup and Setup assignment statements. The cruise control Setup procedure in Figure 47 copies the current value of all local and output signals except timers into the variables that are used during transition execution. This insures that the currently executing transition uses the correct internal state information even though some lower priority transition may have already executed and scheduled changes to that state information.

The cruise control Cleanup procedure in Figure 48 schedules updates of all local and output signals except timers with the current value of the variables. The VHDL simulator updates the signals and Current_State after the specified transition delay. If a lower priority transition has previously scheduled signal updates, they are cancelled by Cleanup's new inertial signal assignment, preempting the lower priority transition. When the ***history-flag*** attribute is "false", we use the transition's *to-state* attribute and Superstate *initial-state*

```
procedure Setup is
begin
 T_Idle_Start_Time                := Idle_Start_Time;
 T_Speed_Change_Start_Time        := Speed_Change_Start_Time;
 T_Acceleration_Change_Start_Time := Acceleration_Change_Start_Time;
 T_Start_Cruising_Start_Time      := Start_Cruising_Start_Time;
 T_Start_Accelerating_Start_Time  := Start_Accelerating_Start_Time;
 T_Stop_Accelerating_Start_Time   := Stop_Accelerating_Start_Time;
 T_Turn_Off_Start_Time            := Turn_Off_Start_Time;
 T_Target_Speed                   := Target_Speed;
 T_Acceleration_Voltage           := Acceleration_Voltage;
 T_Cruise_Voltage                 := Cruise_Voltage;
 T_Old_Speed                      := Old_Speed;
 T_Old_Acceleration               := Old_Acceleration;
 T_Actuator_Voltage               := Actuator_Voltage;
end Setup;
```

Figure 47. VHDL Cruise Control Setup Procedure

attributes to determine which state to pass in for Next_State. When a transition's *history-flag* attribute is "true", we pass the current value of Current_State to Cleanup, "returning" the FSM to the current state, correctly implementing Reacto history transitions.

Timers are not included in the Setup and Cleanup procedures because there are distinct differences between the way timer assignments are handled in VHDL and Reacto. The Lift R-Spec statement:

```
Timeout-Timer <- Clock;
```

is translated to the V-Spec statement:

```
Timeout_Timer <= Max_Time_Reset, Now after Timeout_Timer_Duration;
```

The VHDL statement resets the timer in the next delta delay cycle with Max_Time_Reset, and schedules the timer to change to the current time (Now) Timeout_Timer_Duration milliseconds in the future. This accomplishes the following functions:

- Immediately cancels pending timer assignments.

```
procedure Cleanup (Next_State : State_Type;
                   D          : time) is
begin
Idle_Start_Time                <= T_Idle_Start_Time after D;
Speed_Change_Start_Time        <= T_Speed_Change_Start_Time after D;
Acceleration_Change_Start_Time <= T_Acceleration_Change_Start_Time
                                  after D;
Start_Cruising_Start_Time      <= T_Start_Cruising_Start_Time after D;
Start_Accelerating_Start_Time  <= T_Start_Accelerating_Start_Time
                                  after D;
Stop_Accelerating_Start_Time   <= T_Stop_Accelerating_Start_Time
                                  after D;
Turn_Off_Start_Time            <= T_Turn_Off_Start_Time after D;
Target_Speed                   <= T_Target_Speed after D;
Acceleration_Voltage           <= T_Acceleration_Voltage after D;
Cruise_Voltage                 <= T_Cruise_Voltage after D;
Old_Speed                      <= T_Old_Speed after D;
Old_Acceleration               <= T_Old_Acceleration after D;
Actuator_Voltage               <= T_Actuator_Voltage after D;
Current_State                  <= Next_State after D;
end Cleanup;
```

Figure 48. VHDL Cruise Control Cleanup Procedure

- Makes the predicate of the Timeout transition false in the next delta-delay cycle. (i.e., the timer is reset if it has gone off.)
- Sets Timeout_Timer to go off in the future (unless it is reset before Timeout_Timer_Duration elapses), which will cause the FSM process to execute Timeout_Timer_Duration milliseconds in the future, because Timeout_Timer is in its sensitivity list.
- When Timeout_Timer goes off, it makes the Timeout transition predicate true if we're still in the stopped state.

These functions insure that V-Spec timer behaves the same as the R-Spec timer, but the fact that two values are scheduled for the timer in VHDL keeps us from including timers in the Setup and Cleanup procedures. In fact, we do not declare timer variables in the process declarations since timers are not included in the Setup and Cleanup procedures.

Because timers are a special case, specifiers must take care to insure timers are set and reset explicitly by transitions subject to response-response constraints.

5.4 Generating VHDL Functions

The next step is to generate V-Spec functions from R-Spec functions defined in the V-Spec auxiliary files and used by R-Spec transitions. Figure 49 is the V-Spec version of the R-Spec function transformed from Chapter IV, Figure 39. This transformation is straight forward, because the semantics of Refine's if-then-else is the same as VHDL's.

We do not transform the R-Spec get-input function to VHDL. Get-input's purpose is to drive FSM inputs, and report FSM outputs in the Reacto-Emacs window during simulation as discussed in Section 4.2.2. In VHDL, the simulator and *Testbench* entity drive input and report output replacing the get-input function. We describe *Testbench* creation in Section 6.1.

In addition to generating functions which are defined in the R-Spec, we must generate functions to perform operations which are defined in Reacto, Refine and lisp but not in VHDL. Simple examples include the refine operator implies and the lisp operator Min. Figure 50 shows our VHDL functions for these two operations.

```
function Get_Cruise_Voltage (Target : real;
                             Actual : real) return real is
   variable Delta_Speed : real := Target - Actual;
   variable Voltage     : real := 0.0;
begin
   if Delta_Speed > 2.0 then
      Voltage := 8.0;
   elsif Delta_Speed >= -2.0 and Delta_Speed <= 2.0 then
      Voltage := 2.0 * (Delta_Speed + 2.0);
   else
      Voltage := 0.0;
   end if;
   return Voltage;
end Get_Cruise_Voltage;
```

Figure 49. VHDL Get_Cruise_Voltage Function

```
function Min (left, right : real) return real is
begin -- Min
   if left <= right then
      return left;
   else
      return right;
   end if;
   end Min;

function Implies(A, B : boolean) return boolean is
begin -- Implies
   if not A then
      return true;
   end if;
   return B;
end Implies;
```

Figure 50. VHDL Min and Implies Functions

Our VHDL implies function helps resolve this problem, but it does not provide a "short-circuit" implies, hence schedule lifts' R-Spec state assertion is transformed into several VHDL statements as shown in Figure 51. The if statement condition avoids a

```
assertion Num-Summons > 0 =>
          Total-Response-Time / Num-Summons <= Average-Response-Limit
```

Lift R-Spec State Assertion

```
if Current_State'active and Num_Summons > 0 then
   assert Implies(Num_Summons > 0,
                  Total_Response_Time / Num_Summons <=
                  Average_Response_Limit)
      report "Schedule_Lifts State assertion failed"
      severity warning;
end if;
```

Lift V-Spec State Assertion

Figure 51. VHDL Implies Problem

divide-by-zero error that occurs because Num_Summons is initially zero.

Additionally, sets and set operations are not available in VHDL. Solving this problem requires generating a set data type and operations on those sets. Since we use the set data type in both the lift and schedule lifts FSMs, and their associated Testbenches, we define an integer set data type and the traditional set operations in a separate package *Integer_Sets*. Volume II includes our implementation.

One of the set operations we implement is the Refine "arb" operator. Because the behavior of Refine's arb operator is not spelled out explicitly in the *Refine Users Guide* (30), our implementation probably does not select the same element from a set as the Reacto arb operator, leading to potential differences in the simulations. For example, the Refine arb may pick lift 1 to handle some request, and our VHDL arb operator may pick lift 3, leading to divergent test results from that point on.

We map R-Spec sets of non-integer type to sets of integers by providing map (M) and reverse-map (RM) functions in our *Lift_Types* package (see Volume II).

Whenever the R-Spec uses quantification, we write a special function to iteratively perform the quantification and include it in the V-Spec functions. Figure 52 shows an example R-Spec quantification statement and the corresponding V-Spec function. Both the Reacto quantification statement and the VHDL function evaluate to true when there is at least one lift not in the Off state.

```
ex(Lifts : Status_Type)(Lifts in Status and Lifts.State /= Off)
```

Reacto Quantification Statement

```
function Ex_Not_Off (Status : Status_List) return boolean is
   variable Temp : boolean := false;
begin
   for Lift in 1 to Num_Lifts loop
      if Status(Lift).State /= Off then
         Temp := true;
      end if;
   end loop;
   return Temp;
end Ex_Not_Off;
```

Corresponding VHDL Quantification Function

Figure 52. Set Quantification Example

5.5 Generating Assertion Procedures

R-Spec assertions are the key to verifying R-Spec behavior and consistency. We use them to verify our V-Spec by transforming them into V-Spec assertion procedures as depicted in Figure 53. Recognize that lines 4 through 7 are almost identical to the R-Spec Idle state assertion. We restrict V-Spec assertion verification with the *if current-state'active then* clause (line 3). This means we check assertions after every transition execution, but not during delta-delay cycles or in conjunction with simulation cycles triggered by

```
the-state IDLE
  assertion ACTUATOR-VOLTAGE = 0.0 &
            CLOCK - IDLE-START-TIME <= IDLE-TIME-LIMIT
```

Reacto Cruise Control Idle Assertion

```
1 procedure Idle_Assertion  is
2 begin -- Idle_Assertion
3   if Current_State'active then
4      assert Actuator_Voltage = 0.0 and
5             now - Idle_Start_Time <= Idle_Time_Limit
6        report "Idle State assertion failed"
7        severity warning;
8      assert Implies(Idle_Start_Time /= Start_Time_Reset,
9                     In_Top_Gear'last_event <= Idle_Time_Limit or
10                    Braking'last_event <= Idle_Time_Limit)
11       report "VHDL 'last_event Idle State assertion failed"
12       severity warning;
13  end if;
14 end Idle_Assertion;
```

VHDL Cruise Control Idle Assertion Procedure

Figure 53. Assertion Transformation Example

asynchronous input events. If we attempt to verify assertions during asynchronous event cycles, temporal assertions fail because start time logs checked after the previous transition are not reset until the next transition executes.

The translation process does not generate lines 8 through 12; we add Signal'last-event assertion statements like

```
assert Implies(Idle_Start_Time /= Start_Time_Reset,
               In_Top_Gear'last_event <= Idle_Time_Limit or
               Braking'last_event <= Idle_Time_Limit)
   report "VHDL 'last_event Idle State assertion failed"
   severity warning;
```

to investigate the effects of asynchronous events and multiple synchronous events on the FSM. In Section 4.1.1 we explain that we use *start time logs* to "mark the time that events occur", but actually start time logs only tell us the time that we start responding to a particular event. Generally, we respond to an event when it occurs, but for asynchronous events and multiple synchronous events, this may not be the case. We must examine event history data to determine when the asynchronous events and multiple synchronous events actually occur. Signal'last-event assertion statements are not part of the R-Spec, but they can be added to the R-Spec if we expand our Reacto get-input function to log the last-event times. Instead, we demonstrate the power of looking at event history information in VHDL since the capability to look at signal history is already implemented via the signal'last_event attribute. In this case we know which events cause the Idle_Start_Time start time log to be set, and we examine In_Top_Gear and Braking event histories to determine if we have actually met the Idle_Time_Limit constraint.

Usually, we only examine event histories of input signals, but since we define internal variables as signals in the V-Spec we can examine their histories also. We discuss one such situation involving Current_State in Section 6.5.2.

5.6 Generating Transition Procedures

We generate V-Spec transition procedures from each R-Spec transition object's action attribute except for the Wait transition. As described in Section 4.1.1, wait's functions

are to get input and update the system clock. Since the VHDL simulator assumes the functions of managing the system clock and driving the V-Spec with input, we do not need the wait transition in the V-Spec. Figure 54 repeats the Reacto source code for the cruise control's Startup transition depicted earlier in Figure 31. Figure 55 is the Reacto source

```
1 "Startup"
2 a-transition off-transition-1 from Off to On
3  predicate Activate & Current-Speed >= the-real(30.0)
4              & In-Top-Gear & ~Braking & Engine-Running & ~Deactivate
5  action Start-Cruising-Start-Time <- Clock;
6         Turn-Off-Start-Time <- Start-Time-Reset;
7         Clock <- Clock + Startup-Delay;
8         Old-Speed <- Current-Speed;
9         Target-Speed <- Current-Speed;
10        Cruise-Voltage <- Get-Cruise-Voltage(Target-Speed,
11                                             Current-Speed);
12        Actuator-Voltage <-
13           Cruise-Voltage
14             Min (Max-Volt-Change-Per-ms * Startup-Delay);
15        Get-Input()
16  priority 20
```

Figure 54. Reacto Cruise Control Startup Transition Source Code

code for the cruise control's Startup transition transformed from the Reacto transition depicted in Figure 54. As mentioned in Section 5.2, transitions execute in three steps. First, the transition calls procedure Setup. Second, the transition executes the V-Spec statements transformed from the R-Spec transition action. And third, the transition calls Cleanup. The R-Spec transition action statements in Figure 54 lines 5-15 map directly to the V-Spec statements in Figure 55 lines 4-12, with the exception of the Clock update statement in R-Spec line 7 and the get-input call in line 15. The *Startup-Delay* in R-Spec line 7 is passed to the Cleanup procedure in V-Spec line 13, scheduling the changes made by Startup to occur Startup-Delay in the future. The get-input call in line 15 is not transformed to VHDL, because the VHDL testbench and simulator provide V-Spec input.

```
1 procedure Startup  is
2 begin -- Startup
3    Setup;
4    T_Start_Cruising_Start_Time := now;
5    T_Turn_Off_Start_Time := Start_Time_Reset;
6    T_Old_Speed := Current_Speed;
7    T_Target_Speed := Current_Speed;
8    T_Cruise_Voltage := Get_Cruise_Voltage(T_Target_Speed,
9                                           Current_Speed);
10   T_Actuator_Voltage :=
11      min(T_Cruise_Voltage,
12          Max_Volt_Change_Per_ms * real(Startup_Delay / TC));
13   Cleanup(Cruise, Startup_Delay);
14 end Startup;
```

Figure 55. VHDL Cruise Control Startup Transition Procedure

5.7 Generating the FSM Process Body

Finally, we describe generating the FSM Process Body. We use the following R-Spec attributes to generate the FSM process body:

- State Attributes
  - substates
- Transition Attributes
  - from-state
  - predicate
  - priority

Figure 56 shows the general form of the FSM process body using a portion of the cruise control FSM process body. The line numbers are added for the purposes of discussion.

The process body consists of a simple if statement and a controlling case statement. The if statement (lines 2-4) resets the variable Scheduled-Priority when the VHDL simulator completes a scheduled transition execution. This allows any transition whose predicate

```
 1 begin
 2   if Current_State'active then
 3      Scheduled_Priority := integer'low;
 4   end if;
 5   case Current_State is
 6      when Off =>
 7         Off_Assertion;
 8         ...
 9      when Idle =>
10         On_Assertion;
11         Idle_Assertion;
12         if not Engine_Running or Deactivate and
13            100 > Scheduled_Priority then
14            Shutdown;
15            Scheduled_Priority := 100;
16         elsif Resume and In_Top_Gear and not Braking and
17               20 > Scheduled_Priority then
18            Resume_And_Enabled;
19            Scheduled_Priority := 20;
20         elsif Start_Acceleration and In_Top_Gear and not Braking and
21               15 > Scheduled_Priority then
22            Accelerate_And_Enabled;
23            Scheduled_Priority := 15;
24         end if;
25      when Accelerating =>
26         ...
27      when Cruise =>
28         ...
29   end case;
30 end process Cruise_Control;
```

Figure 56. VHDL FSM Process Body

is true to execute subject to the case statement. The case statement (lines 5-29) controls the V-Spec FSM. It maintains the current state and calls the assertion and transition procedures. There is a case statement option (lines 6, 9, 25, and 27) for each primitive state in the FSM.

Within each option, the FSM executes applicable assertion procedures[1] first (lines 7, 10, 11). If any assertion is false, the simulator outputs a warning message, and continues executing. For example, we call the assertion procedure for the On state and the Idle state (lines 10 and 11) because the Idle state is a substate of On.

Second, the FSM evaluates applicable transition predicates in an if-elsif hierarchy (lines 12-24). When the cruise control is Idle, transitions Shutdown (line 14), Resume_And_Enabled (line 18), and Accelerate_And_Enabled(line 22) are all applicable. It evaluates transition predicates in priority order, executing the first transition with a true predicate if that transition's priority is greater than any currently scheduled priority. This effectively implements the PEM discussed earlier in Section 5.1. Note that no transition preempts itself, since no transition priority is greater than itself.

It is important to specify a distinct priority for each transition to avoid creating a different V-Spec priority scheme than the R-Spec as discussed in Section 4.2.2.

5.8 Automating The Transformation

At the beginning of this chapter, we specify a general mapping from Reacto to VHDL, and in the previous paragraphs we describe a detailed transformation process from the R-Spec to the V-Spec. Manually performing the transformation from the R-Spec source files is tedious. Even minor errors can cause big problems. We can automate most of the transformation process, significantly improving its speed and accuracy.

The automated transformation can use the R-Spec source files as input or simply use the R-Spec in the knowledge base as an abstract syntax tree, producing the V-Spec source code directly from it.

[1] Assertion and transition procedures are applicable if they are associated with the primitive state or a superstate of the primitive state as determined from the states *substates* attribute.

There are some complications to consider in order to automate the transformation process. For example, generating the V-Spec entity declaration requires knowing which variables are the inputs and outputs. The R-Spec inputs and outputs are identified by comments in the auxiliary file and such comments are not currently present in Refine's knowledge base. The entity declarations could be generated from the SADT diagram using Eickmeier's method (12:64-65,123-124), or we could use Reacto's interface variables to specify our inputs and outputs if they evolve to something like VHDL signals in future versions of Reacto. In any case, adding a Reacto graphical interface specification capability ala SADT to Reacto would make it easier to specify inputs and outputs and understand the state machine in its environment.

Current differences in the Reacto and VHDL languages also complicate automated transformation. Generally, Reacto is more abstract than VHDL, therefore we generate a lot of VHDL code to support Reacto operators, sets and set operations as discussed in the previous sections. These problems are not limited to those already mentioned; for example, if we need to use Reacto maps, we would have to generate even more VHDL code.

A second difference between Reacto and VHDL is that VHDL is more strongly typed than Reacto. This causes some difficulty mapping from Reacto symbol types to VHDL enumerated types as shown in Section 5.2. Strong typing also causes problems with expressions of type time. In Reacto, expressions involving integer time types work correctly with real data types, but VHDL's strong typing forces us to add a conversion factor and explicit type conversion. Figure 57 shows an example Reacto statement and corresponding VHDL statement illustrating this problem. We declare the Time constant *TC* in Figure 57 explicitly for the purpose of mixing time type with real in this expression.

These and other problems must be successfully addressed to completely automate the transformation process, but even a partially automated transformation could save considerable time and reduce errors.

```
Max-Volt-Change-Per-ms * Startup-Delay
```

Reacto Expression

```
constant TC : Time := 1 ms; -- Time conversion factor
...
Max_Volt_Change_Per_ms * real(Startup_Delay / TC)
```

Corresponding VHDL Expression

Figure 57. VHDL Strong Typing Example

5.9 Conclusion

In this chapter we describe translating from Reacto to VHDL. The Reacto to VHDL transformation is a significant portion of our research. We define the transformation process by applying it to our example problems. The transformation process exposes the differences between the languages, and emphasizes the difficulties of describing the relationships of system inputs and outputs with respect to time in the two languages. We encourage the use of an automated transformation process to eliminate errors and expedite the transformation.

Our three example transformations do not constitute a rigorous and formal mapping between Reacto and VHDL. They are specific examples of behavior-preserving transformations. Defining the rigorous mapping is essential before automating the transformation from Reacto to VHDL.

In order to reconcile the differences in the way our Reacto model and VHDL control the clock and drive simulation input, we introduce a *Preemptive Execution Model* (PEM) for VHDL. The PEM allows higher priority transitions to preempt lower priority transitions, giving us the ability to investigate the effects of asynchronous events shedding light on transition dependencies.

VI. Applying VHDL

In the previous Chapters we discuss augmenting Reacto with time, applying Reacto to our two problems, and transforming the R-Specs in V-Specs, now we relate applying VHDL to our example problems by discussing the following items:

- Driving the VHDL Simulation
- Increasing Simulation Power
- Running the VHDL Simulator
- VHDL Test Cases
- VHDL Specification Improvements

6.1 Driving the VHDL Simulation

Successful application of the steps defined in Section V provides us with a V-Spec. Before we can execute the V-Spec, we must create a separate entity called a *Testbench.* We use the Testbench to drive FSM input and to evaluate and report FSM output. After we generate the Testbench, we create a VHDL configuration connecting the Testbench to the V-Spec FSM and execute the V-Spec.

6.1.1 Testbench Generation Since we do not transform the Testbench from the R-Spec, we now describe generating a Testbench. Volume II contains the VHDL code for the cruise control Testbenches. A Testbench has two parts, an entity declaration, and an architecture containing the behavioral description of the Testbench. Our Testbenches are null entities like the one defined in Figure 58.

```
entity Cruise_Test is
end Cruise_Test;
```

Figure 58. VHDL Null Cruise_Test Entity

The VHDL architecture we define for the null Testbench entity has two sections, declarations and body. In the architecture declarations section, we declare the components hooked to the Testbench and the signals we need to connect the Testbench to the components. We may have more than one component to connect. If, for example, we connect our lift and schedule-lifts FSMs together for a test, they would each be declared in the architecture declarations section.

Our Testbench architecture body has three parts, component instantiations, driver process, and monitor processes. The component instantiations name the previously declared components, and provide the opportunity to create multiple instantiations of each component. If for example, we want to include multiple lifts in our simulation, we instantiate them, specifying their connections to the declared signals via a VHDL generate statement as shown in Figure 59. Collectively, we refer to the instantiated components as the Test System (TS).

```
Lifts : for i in 1 to Num_Lifts generate
 L : A_Lift
      generic map (i)
      port map (
         Dest_Buttons(i),
         Emergency_Buttons(i),
         Outstanding_Requests,
         Summons(i),
         Dest_Lights(i),
         Status(i));
 end generate;
```

Figure 59. VHDL All Lift Instantiation

The next part of the Testbench is the driver process. The driver process drives the inputs of the TS by assigning new values to the signals connected to the TS. The driver process runs concurrently with the TS, and we use wait statements to coordinate the Testbench with the TS as desired during the simulation. Because the Testbench and the TS run concurrently, we are not constrained to synchronize inputs with FSM transitions as we are in our Reacto simulation. We implement the R-Spec test cases by creating VHDL

signal assignment statements and wait statements that provide the same input to the TS. We write VHDL assertion statements in the driver process to examine TS outputs for expected behavior and notify us of anomalies during simulation.

The third part of the Testbench is the monitor process section. The monitor processes provide simulation output to the simulator window so we can examine Testbench and TS behavior. We create a monitor process for each signal used to connect components and the Testbench. As events occur on a signal, its monitor process writes the time, signal name and signal value to the simulation window, giving us a log of system behavior.

6.1.2 Testbench Configuration The final step before VHDL simulation is to generate the test configuration. The test configuration simply identifies which library components we wish to connect into the Testbench architecture. The All_Lift_Test configuration, Test2, shown in Figure 60 is a good example. *Schedule_Lifts(Reacto)* in Figure 60 is the

```
configuration Test2 of All_Lift_Test is
  for Behavior
      for all: Schedule_Lifts
         use entity work.Schedule_Lifts(Reacto);
      end for;
      for Lifts
         for all: A_Lift
             use configuration work.Motorized_Lift;
         end for;
      end for;
   end for;
end Test2;
```

Figure 60. Lift Configuration

V-Spec architecture, and *Motorized_Lift* is another configuration including the V-Spec lift and the VHDL entity *Motor*. Motor models the floor sensor input to a lift based on the lift's motor control output, freeing us from having to generate floor sensor input to the lifts from the driver process. We use this example to illustrate increasing simulation power in the next section.

6.2 *Increasing Simulation Power*

In Reacto, we are constrained to simulate one R-Spec at a time, and to provide all of the inputs to that R-Spec interactively or through the simulation script file. Because VHDL is a concurrent language, we can simulate more than one entity simultaneously. This allows us to simulate multiple FSMs and to model other entities in conjunction with FSMs. This capability increases simulation power tremendously. We use our lift simulations to illustrate this power.

The SADT diagram in Figure 61 shows how we connect a "motor" to the lift FSM, relieving us of the responsibility to provide floor sensor input to lifts from the Testbench. We

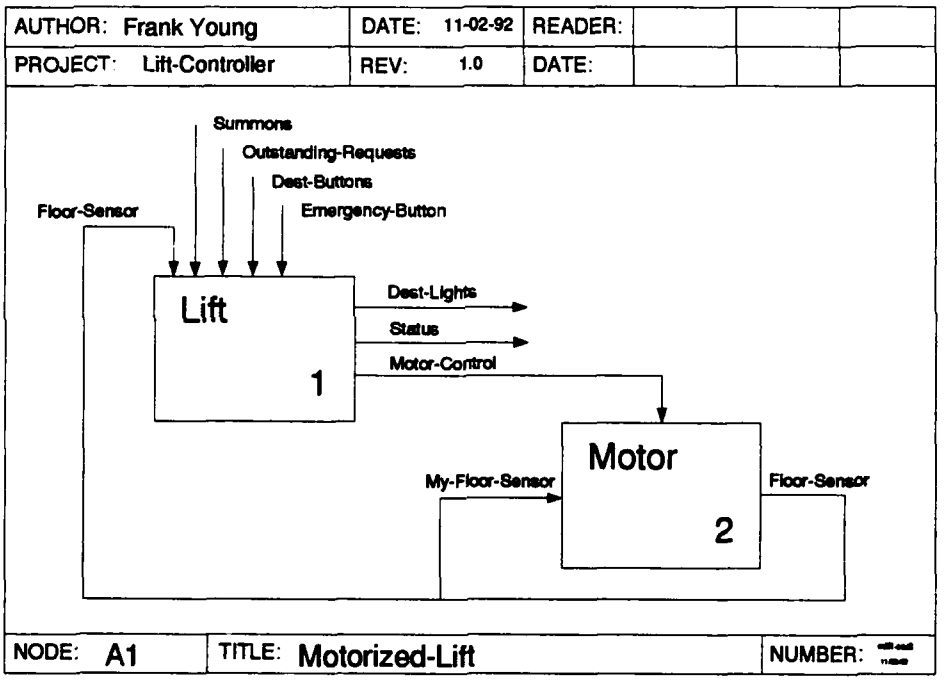

Figure 61. Motorized Lift SADT Diagram

specify Motor behavior in the file *motor.vhd*, included in Volume II. We write conventional VHDL describing the motor architecture behavior, but we could create an R-Spec FSM to specify motor behavior and transform the Motor to VHDL as described in Section V.

Now that we have a motor for our lift, we declare a motorized lift entity and connect the motor and lift in a configuration. Now we can create instantiations of motorized lifts without configuring motors and lifts together every time we want to use one. We build the

configuration *Motorized_Lift* in file *motorized_lift.vhd*, also contained in Volume II. Now, no matter how many lifts we want to simulate, we do not have to generate any floor sensor inputs. We also create monitor processes in the Motorized lift configuration architecture. They provide simulation output for all lift inputs and outputs including motor control and floor sensor information. We use a VHDL generic to provide a unique number for each lift, and include that lift number in all lift output during the simulation so we can associate output with the correct lift.

The motorized lift example illustrates our ability to model other objects in conjunction with the objects we are specifying. This increases the power of the simulation by reducing the amount of simulation input we must generate in the Testbench. The behavior of these objects can be defined by FSMs or any algorithm or structure (a composition of entities).

In the lift controller problem, this allows us to easily vary the number of lifts in the simulation, just by changing the constant parameter *Num_lifts* in package Lift_Types. We use the *Big_Test* configuration and architecture to simulate nine lifts and 20 floors with the motorized_lift configuration. Coordinating all 9 floor sensor inputs in the Testbench without the motorized lift configuration presents an apparently intractable problem, but VHDL's simulation power allows us to overcome it easily.

6.3 Running the VHDL Simulator

There are three ways to run the simulations in VHDL. The first way is through the interactive simulator *VHDLSIM*. VHDLSIM provides a command line interface to the Testbench. The second way is through the VHDL debugger *VHDLDBX*. VHDLDBX also provides a command line interface, but in addition, it provides a user friendly mouse interface and scrolling windows to control and watch the simulation. The third and final way is to use a script file, sending output directly to a system text file for later analysis and to archive the test results.

We use VHDLDBX to debug and understand V-Spec execution. We use the script file method to archive and compare simulation results with the R-Spec simulations. As we

analyze simulation results, we observe behavior problems and make improvements to the specification. In the next sections, we discuss test cases used, and resulting improvements.

6.4 VHDL Test Cases

Generally, the VHDL test cases are the same as the Reacto test cases described in Section 4.3. Some test cases are new, and we describe those new test cases here. Our final V-Specs pass all of these tests, outputting the correct results without failing any assertions. We refer the reader to Volume II of this thesis for actual VHDL input testbench files and results of each test case.

6.4.1 Cruise Control Test Cases Except for the asynchronous event tests, all VHDL cruise control test cases are identical to the Reacto test cases. We show the cruise control input to and output from the Activation Allowed Test as we do for Reacto in Section 4.3.1.

Initialization Test (Same as Reacto)

Activation Denied Test (Same as Reacto)

Activation Allowed Test Same as Reacto, but explained here for comparison. This tests that the cruise control shall activate when the engine is running, the transmission is in top gear, and the speed is at least 30 miles per hour (mph). Requirements Tested: R1, R2, R3, C4. Test steps:

1. Set In-Top-Gear, Engine-Running to true
2. Set Braking to false
3. Set Speed to 30 mph
4. Toggle Activate between true and false
5. Set Speed to 29 mph
6. Is Actuator-Voltage greater than 0.0 volts?

Actual test input is shown in Figure 62. Note that VHDL inputs are compiled VHDL statements in the VHDL testbench, not strings in an input text file like our

Reacto input files. The assertion statement checks the output of the cruise control for us during the simulation, and if the output is not what we expect, it outputs the error message "Test Error." The two wait statements coordinate the timing of the test inputs with the TS without concern for actual transition delays. The Shut_Off procedure call turns the cruise control off in preparation for the next test.

```
My_IO.Put_Line ("Start Activation Allowed Test ****");
In_Top_Gear          <= true;
Braking              <= false;
Current_Speed        <= 30.0;
Engine_Running       <= true;
Activate             <= true, false after 1 sec;
wait for 10 sec;
Current_Speed        <= 29.0;
wait for 1 sec;
assert Actuator_Voltage > 0.0
report "Test Error"
severity warning;
wait for 10 sec;
Shut_Off;
My_IO.Put_Line ("Stop Activation Allowed Test ****");
```

Figure 62. VHDL Cruise Control Activation Allowed Input

Actual *Activation Allowed Test* output is shown in Figure 63. Note that all input and output events are logged along with the time of each event. The reason there is no event for the *Set Braking to false* step, is that Braking is already false because it is changed by the Shut_Off procedure during the previous test. The word "event" is output in conjunction with the Actuator_Voltage value, indicating that the cruise control changed the value of Actuator_Voltage from its previous value. In other tests, the word "Transaction" indicates reassignment of the old value, and the word "Trigger" indicates that we *triggered* the Actuator_Voltage monitor process by changing the boolean signal *Trigger* to output the current value of Actuator_Voltage without any assignment to Actuator_Voltage.

```
Start Activation Allowed Test ****
   6000 MS In_Top_Gear          : TRUE
   6000 MS Current_Speed        : 30.00
   6000 MS Engine_Running       : TRUE
   6000 MS Activate             : TRUE
   6250 MS Actuator_Voltage     : Event 0.20
   7000 MS Activate             : FALSE
   7250 MS Actuator_Voltage     : Event 1.00
   8250 MS Actuator_Voltage     : Event 1.80
   9250 MS Actuator_Voltage     : Event 2.60
  10250 MS Actuator_Voltage     : Event 3.40
  11250 MS Actuator_Voltage     : Event 4.00
  16000 MS Current_Speed        : 29.00
  17000 MS Actuator_Voltage     : Event 4.80
  18000 MS Actuator_Voltage     : Event 5.60
  19000 MS Actuator_Voltage     : Event 6.00
  27000 MS In_Top_Gear          : FALSE
  27000 MS Braking              : TRUE
  27000 MS Deactivate           : TRUE
  27000 MS Stop_Acceleration    : TRUE
  27000 MS Current_Speed        : 0.00
  27000 MS Engine_Running       : FALSE
  27500 MS Actuator_Voltage     : Event 0.00
  28000 MS Braking              : FALSE
  28000 MS Deactivate           : FALSE
  28000 MS Stop_Acceleration    : FALSE
Stop Activation Allowed Test ****
```

Figure 63. VHDL Cruise Control Activation Allowed Output

At 6000 ms, we tell the cruise control to activate. Since the transmission is in top gear, the speed is at least 30 miles per hour, the engine is running, and the brakes are off, the cruise control turns on. It sets the output voltage to 0.20 volts 250 ms later at 6250 ms. Voltage does not increase to 4.0 volts immediately because this would violate the maximum allowed voltage increase of 0.8 volts/second. Instead, at one second intervals, cruise control increases the voltage by the amount allowed until it reaches 4.0 volts. At 16000 ms, we change the current speed to 29 miles per hour, and the cruise control responds by raising the voltage to increase the speed subject to the 0.8 volts/second constraint. At 19000 ms voltage reaches the required 6.0 volts. At 27000 ms, we shut the cruise control off in preparation for the next test. These are the expected results, and no VHDL assertions are violated, so this version of the cruise control passes the *Activation Allowed Test.* We compare this output with the Reacto output from the same test in Section 7.1.

Deactivation Test (Same as Reacto)

Acceleration Test (Same as Reacto)

Resume Test (Same as Reacto)

Downhill Test (Same as Reacto)

Uphill Test (Same as Reacto)

Breaking During Activate Delay Test is an asynchronous event test, meant for use with VHDL, and explicitly designed to expose timing dependencies between the Startup and Not-Enabled transitions. It models the situation where the driver activates the cruise control, and less than 250 ms later puts on the brakes.

Figure 64 is the output from the VHDL Breaking During Activate Delay Test, and as you can see, we are able to input a Braking event at 228100 ms, independent of the cruise control's output events at 228250 ms or 228500 ms and the Activate events at 228000 ms and 229000 ms. We are constrained to do differently in the Reacto test depicted in Figure 34. Here, the cruise control output voltage is 0.00 volts at 228500 ms, meeting the time constraint to stop cruising within 500 ms of the Braking

```
Start Breaking During Activate Delay Test ****
 228000 MS In_Top_Gear         : TRUE
 228000 MS Current_Speed       : 45.00
 228000 MS Engine_Running      : TRUE
 228000 MS Activate            : TRUE
 228100 MS Braking             : TRUE
 228250 MS Actuator_Voltage    : Event 0.20
 228500 MS Actuator_Voltage    : Event 0.00
 229000 MS Activate            : FALSE
 233000 MS Braking             : FALSE
 239250 MS In_Top_Gear         : FALSE
 239250 MS Braking             : TRUE
 239250 MS Deactivate          : TRUE
 239250 MS Stop_Acceleration   : TRUE
 239250 MS Current_Speed       : 0.00
 239250 MS Engine_Running      : FALSE
 239750 MS Actuator_Voltage    : Transaction 0.00
 240250 MS Braking             : FALSE
 240250 MS Deactivate          : FALSE
 240250 MS Stop_Acceleration   : FALSE
Stop Breaking During Activate Delay Test ****
```

Figure 64. VHDL Cruise Control Breaking During Activate Delay Test Output

event at 228100 ms, passing the test. We turn the cruise control off at 239250 ms, preparing for the next test.

Breaking During Activate Asserted Test is an asynchronous event test, meant for use with VHDL, and explicitly designed to expose timing dependencies between the Startup and Not-Enabled transitions. It is similar to the Breaking During Activate Delay Test, except for the fact that the braking event occurs more than 250 ms after the activate event, while activate is still asserted. We expect the cruise control to turn off within 500 ms of the braking event.

Breaking After Activate B4 Cruise Test is an asynchronous event test, meant for use with VHDL, and explicitly designed to expose timing dependencies between the Startup and Not-Enabled transitions. Similar to the two previous test cases, this test case examines the behavior of the cruise control when the driver presses and releases the activate button very quickly and puts on the brakes within 250 ms of pushing the activate button. Again, we expect the cruise control to output zero volts within 500 ms of the braking event.

Resume During Breaking Test is an asynchronous event test, meant for use with VHDL, and explicitly designed to expose timing dependencies between the Resume-And-Enabled and Not-Enabled transitions. It models the situation where the driver activates the cruise control, puts on the brakes after the cruise control activates, and while the brakes are pressed, pushes the resume button, subsequently releasing the resume button and the brakes simultaneously. We expect the cruise control to continue outputting zero volts within 500 ms of the braking event even though the resume button is pressed.

Deactivate Overlaps Resume Test is an asynchronous event test, meant for use with VHDL, and explicitly designed to expose timing dependencies between the Shutdown , Not-Enabled, and Resume-And-Enabled transitions. It models the situation where the driver activates the cruise control, subsequently deactivates it, and while the deactivate button is pressed and before the cruise control actually shuts down, also pushes the resume button, keeping it pressed until after the deactivate button is

released. We expect the cruise control to continue outputting zero volts within 500 ms of the deactivate event even though the resume button is pressed.

6.4.2 Lift Controller Test Cases Refer to Appendix B, Section B where you can see the details of each test case. In the following Section, we list the lift test cases, and in Section 6.4.2.2, we describe the schedule lifts test cases. As with the cruise control test cases, our final lift controller V-Specs pass these tests, producing the expected results, without VHDL assertion errors.

6.4.2.1 Lift Tests

All Summons Test (Same as Reacto)

All Destinations Test (Same as Reacto)

Emergency Button Test (Same as Reacto)

Mixed Destinations and Summons Test (Same as Reacto)

Timeout Test (Same as Reacto)

6.4.2.2 Schedule Lifts Tests

Off State Test (Same as Reacto)

All-Summons All Lift Test (Same as Reacto All Summons Test)

All Summons 1 Lift Test (Same as Reacto)

All Destinations Test Same as Reacto Lift *All Destinations Test*, except we task all four lifts with all destinations simultaneously.

Two User Lift Test A typical scenario test, 2 users summon lifts, push destination buttons, and travel to the destinations.

Four User Lift Test A typical scenario test, 4 users summon lifts, push destination buttons, and travel to the destinations.

Idle Schedule Test (Same as Reacto)

6.5 VHDL Specification Improvements

Specification improvements we make as a result of VHDL simulation analysis can be categorized into two areas, behavioral and temporal. Behavioral improvements are improvements we make to correct errors in the relationship between FSM inputs and FSM outputs. Temporal improvements are those we make to correct errors in the temporal relationship between input events and output events— i.e., improvements made to correct behavior violating time constraints.

6.5.1 Behavioral Improvements Since the cruise control is a fairly simple problem and we are able to thoroughly debug the cruise control R-Spec, we make no cruise control behavioral improvements as a result of simulating the cruise control V-Spec.

We do however make several behavioral lift specification improvements, mostly because of VHDL's ability to combine FSMs together during simulation, and partly because it is easier to generate more powerful test cases in VHDL. In the first example, we discover that we inadvertently left out setting the Timeout-Timer in the Request-Scheduled transition. If we run the same test case in Reacto, and correctly interpret the results, we see this problem. We discover it in VHDL because VHDL's increased simulation power makes it easier to generate better test cases. Similarly, we discover that we have the priorities reversed on the Lift's Timeout and Resume transitions. We notice it in VHDL rather than Reacto only because of a slightly different test case. We correct the problems in both the Reacto and VHDL models.

When we connect Schedule-Lifts and four lifts together, the VHDL simulation highlights a problem between them. The Request-Scheduled Lift transition moves the lift from Idle to Stopped when Outstanding-Requests includes a summons for the Idle lift's current floor. Originally, the No-Requests Lift transition subsequently returns the lift to the idle state when the lift's floor-set is empty without examining Outstanding-Requests first. Since our Reacto test cases removes the current floor summons from Outstanding-Requests before the Request-Scheduled transition returns the lift to the stopped state, Lift appears to function correctly during R-Spec simulation. However, in VHDL, Schedule-Lifts' Status-Event-Delay is 50 times longer than Lift's Request-Scheduled-Delay plus No-

Requests-Delay delaying the removal of the summons from Outstanding Requests. This causes the lift to toggle back and forth between the Idle and Stopped states 50 times before Schedule-Lifts can update Outstanding-Requests. It is difficult to sort out whether this error is a timing problem between Lift and Schedule-Lifts, or a behavioral problem. Ultimately, we decide it is a behavioral problem. Correcting Lift's No-Requests transition predicate so the lift stays stopped when Outstanding-Requests includes the current floor and direction fixes this incompatibility that is apparent under a VHDL simulation combining the two FSMs together.

6.5.2 Temporal Improvements Examining the effects of asynchronous events on V-Specs sheds light on dependencies between transitions that would not otherwise be noticed. In both the Reacto and VHDL models, the FSM reacts to an event on an input because the new value of that input affects a transition predicate. As a constrained transition reacts to a new input value it sets a start time log, but we must know when the event actually occurred to determine if we meet the constraint when we allow asynchronous events to occur. We look at signal event history with VHDL's signal'last-event attribute in assertions to determine if we meet constraints when we allow asynchronous events in the PEM. If we discover that we fail to meet a time constraint, we must investigate why. We may fail to meet a time constraint for a number of reasons including incorrect priorities, incomplete transition predicates, and dependent transitions. Generally only the incorrect priorities or predicates problem is evident in our Reacto model, but because we can drive signals asynchronously in VHDL, the dependencies are highlighted.

Additionally, because of dependencies in the state machines, multiple events occurring on input signals less than a transition delay apart may be ignored if the state machine is busy responding to an equal or higher priority events. We attempt to insure that critical inputs are not missed by carefully determining transition priorities.

Careful design of test cases, and careful analysis of the simulation output helps resolve missed time constraints and ignored events. For example, assume the cruise control is off, and the user presses activate and subsequently puts on the brakes before the cruise control responds to the activate event. If the user keeps the brakes on, the cruise enters the

cruise state first, then transitions to the idle state. Does the system violate the idle time constraint? If so, the Startup and Not-Enabled transitions are dependent, if not, they are not dependent.

Let us examine this dependency with the cruise control *Braking During Activate Asserted Test.* Figure 65 is a simplified timing diagram of this test. The simultaneous

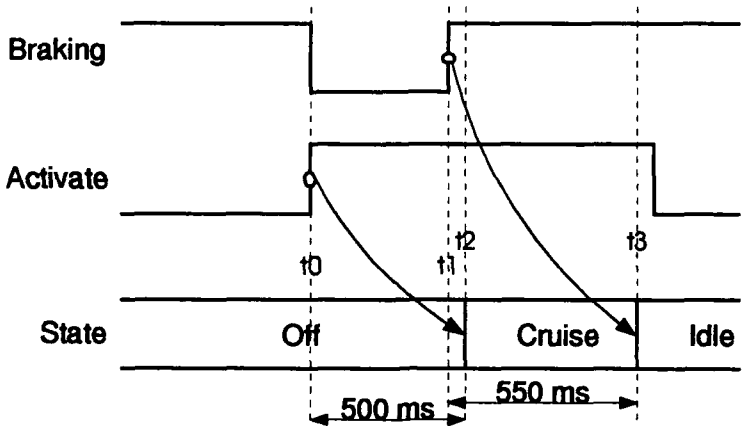

Figure 65. Braking During Activate Asserted Test Timing Diagram

events, activate going true and Braking going false, occur at the beginning of the test at time t0. At time t1, we assert Braking asynchronously. Note that the Braking event does not immediately cause the FSM to take a transition, it depends on the combination of inputs the transition predicates are sensitive to and the new input values. Reacting to the Activate event at time t0, the cruise control enters the Cruise state at time t2 via the Startup transition. Reacting to the Braking event at time t1, the cruise control enters the idle state at time t3 via the Not-Enabled transition. When we examine event history data as the cruise control enters the idle state, the Idle state assertion fails. The text of the assertion is

```
assert Implies(Idle_Start_Time /= Start_Time_Reset,
               In_Top_Gear'last_event <= Idle_Time_Limit or
               Braking'last_event <= Idle_Time_Limit)
   report "VHDL 'last_event Idle State assertion failed"
   severity warning;
```

and it fails because the event on Braking (550 ms) is more than Idle_Time_Limit (500 ms) in the past. This dependency cannot be exposed by an assertion error in our current Reacto model for two reasons. First, our Reacto get-input function does not log event history information for us to evaluate. Second, we cannot inject an asynchronous event in Reacto; we are constrained to inject the Braking event at time t0 (test start time) or at time t2 (at the end of the Startup transition).

We now evaluate that assertion error to determine what it means. The wording of the original specification is such that the Cruise-Control system must be active before the braking or not In-Top-Gear events can cause it to go inactive (17:278). Hence, we define no transition between Cruise-Control's Off and Idle states activated by an event on Braking or In-Top-Gear.[1] We could interpret the informal specification in two ways. One, it means that the time constraint should be measured from the time the state became Cruise to the time the state changed to Idle. If we assume the first interpretation is correct, we can modify the state assertion to allow events on In-Top-Gear or Current-State or Braking within the time limit to meet the constraint as follows:

```
assert Implies(Idle_Start_Time /= Start_Time_Reset,
               In_Top_Gear'last_event <= Idle_Time_Limit or
               Current_State'last_event <= Idle_Time_Limit or
               Braking'last_event <= Idle_Time_Limit)
   report "VHDL 'last_event Idle State assertion failed"
   severity warning;
```

Effectively, this means that there is no dependency between the Startup and Not-Enabled transitions. Second, we can interpret the time constraint to mean the time between the braking or In-Top-Gear event and the state change to Idle. If we assume the second interpretation, then we must reduce the delays of the Startup and Not-Enabled transitions to meet the time constraints because they are dependent. We assume the second interpretation, and set the delays of both transitions to 250 ms, or half of the time constraint each.

[1] Without such a transition, the cruise control stays in the Off state.

This fixes the time constraint violation, because as illustrated in Figure 66 no matter when the Braking event occurs between times t0 and t2, it is at most 500 ms from time t0 to t3.

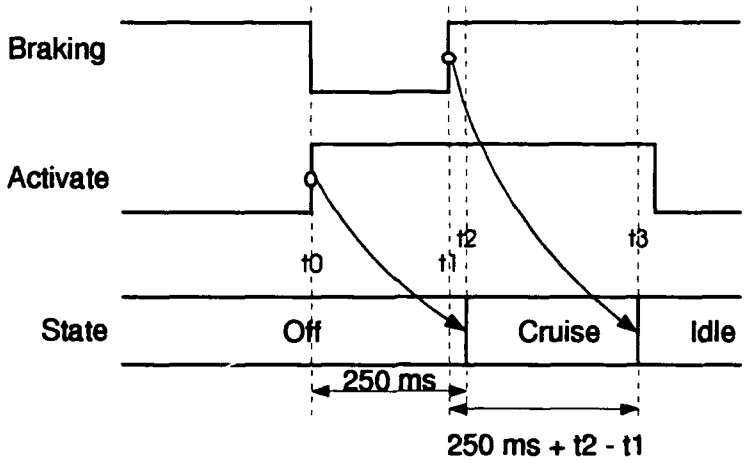

Figure 66. Fixed Braking During Activate Asserted Test Timing Diagram

It is interesting to note that this dependency is not peculiar to our V-Spec; it also applies to a cruise control created using Eickmeier's methodology. Figure 67 is an SADT diagram representing part of the cruise control we developed using his methodology. Referring to Figure 67, each activity has an associated delay. If $E + C + S \leq Idle\_Time\_Limit$ then the delays meet the Idle_Time_Limit time constraint, and when $C + S \leq Start\_Cruising\_Time\_Limit$, the Start_Cruising_Time_Limit constraint is also met. Because the term $C + S$ appears in both formulas, the two constraints are dependent. Setting $E = 250ms$ and $C + S = 250ms$ is equivalent to setting each of our transitions in the V-Spec model to 250 ms.

Injecting asynchronous events into the lift-controller FSMs (Figures 36 and 37) also reveals some dependencies. For example, we discover that the Lifts' New-Destination transition is self-dependent. That is, if multiple events occur on Dest-Buttons less than Light-On-Time-Limit apart, the New-Destination transition cannot meet the Light-On-Time-Limit time constraint for the succeeding events. We fix this dependency by setting New-Destination delay to half its constraint. This solves the problem for this transition under three conditions. One, New-Destination can turn on more than one destination

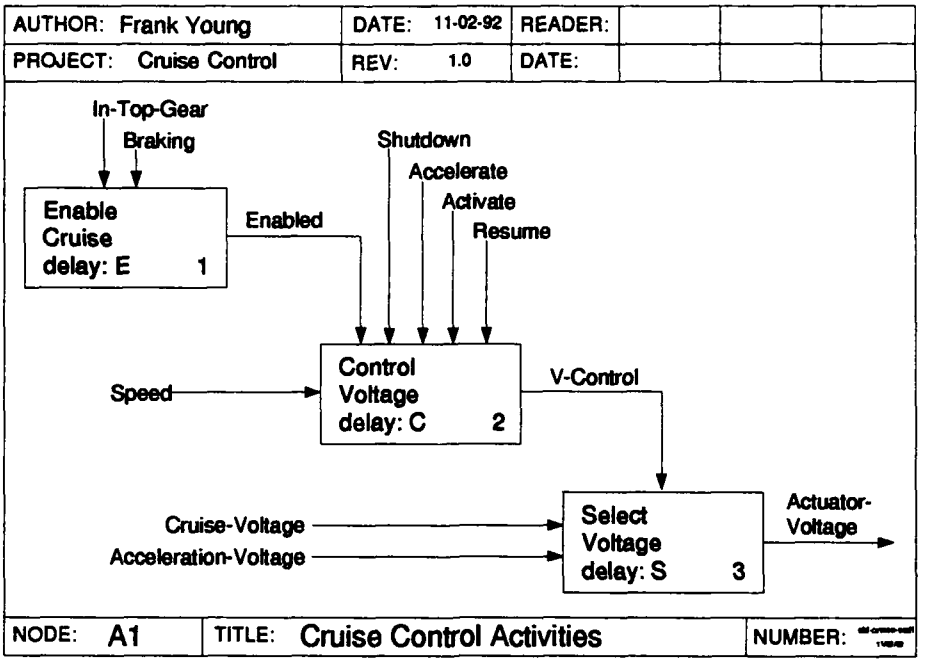

Figure 67. Eickmeier's Methodology Cruise Control SADT Diagram

light simultaneously, handling one or more Dest-Buttons events during each execution. Two, consecutive events on any single Dest-Button occur at least half of Light-On-Time-Limit apart to avoid missing an event[2]. And, three, New-Destination's priority is high enough that it cannot be preempted. Changing New-Destination's delay simply point's out the dependency, leaving the actual implementation up to the system designers. We mention three (of many possible implementations) for resolving this dependency. One, assign dedicated independently operating processors the task of turning on and off the light for each button. Two, assign New-Destination's task to a single independent processor, which can turn on all the lights for all buttons simultaneously in less than half of Light-On-Time-Limit time. Three, assign the New-Destination transition and all higher priority transitions to a processor which accomplishes all activities in less than the shortest time constraint for any of them.

Although we do not include any test case data exposing the dependency, manual simulation in the VHDL debugger exposes a dependency between the Lift's Emergency,

[2]This could be accomplished by electro-mechanical debounced switches, or specified as a separate constraint if it were anticipated that buttons would not remain "pressed" for at least this long.

Not-Emergency, and New-Destination transitions. A sequence of events on an Emergency-Button (causing the FSM to toggle back and forth between the On and Off states) can keep the Lift from responding to Dest-Buttons events for an arbitrary amount of time violating the Light-On-Time-Limit constraint. The dependency between Emergency, Not-Emergency and New-Destination cannot be resolved by manipulating transition delays or transition priorities. We can resolve it by combining these functions together in a sequential implementation (think of it as a single transition and a superstate combining both the Off and On states) capable of meeting the minimum time constraint. Breaking the Reacto Specification down into independent state machines, one for each parallel function, is another option, but this produces more state machines and a less understandable view of the specification. Which way is superior? As long as the implementer understands the need to manage the dependency, either suffices. We leave the specification as is, simply annotating the dependency, leaving the decision about which way to resolve it to the developer. In another circumstance, we might have reason to specify one solution over another, and would change the specification to indicate our preference.

Similarly, while manually exercising the lift in the debugger, we discover a dependency between the Lifts' At-Scheduled-Floor and New-Destination transitions. A sequence of events on the Dest-Buttons can keep the lift from responding to a floor sensor event for an arbitrarily long period of time causing the lift to miss the Light-Off-Time-Limit constraint. As before, we simply note the dependency and leave implementation details to the developer.

A similar dependency exists between Schedule-Lifts' Summons-Button-Event and Status-Event transitions. The Schedule-Lifts' transitions Summons-Button-Event and Status-Change are both time constrained transitions, and they both start and finish in the same state. We observe that simultaneous summons-button and status events can cause both predicates to be true. Since only one transition can execute, the lower priority transition cannot meet it's time constraint. In fact, a sequence of summons-buttons events can delay the execution of the status-change transition for an arbitrary amount of time. We propose three alternatives for eliminating this dependency. One, we could separate these two functions into independent state machines, but but this produces more state machines

and a less understandable view of the specification. Two, if Reacto and our methodology supported orthogonal states, this dependency could be resolved by specifying the On state as an orthogonal state with one substate for each of the transitions operating concurrently like STATEMATE. Alternatively, it could be solved by combining the two transitions into a single transition sensitive to either event and meeting the minimum time constraint. We recognize that these alternatives are implementations that can resolve the dependency. We leave the specification as is, simply pointing out the dependency, not constraining the designer to a particular solution. We do note however, that the STATEMATE orthogonal states approach allows larger and more complex systems to be specified as a single state machine, and that perhaps it points out the dependency just as well. We cannot express orthogonal states in the current R-Spec model, but we could expand the V-Spec to accommodate them by specifying a separate concurrent process for each orthogonal substate and adding its superstate's Current_State to the process sensitivity list.

A sequence of events on Up and Down Buttons can keep Schedule-Lifts from responding to Status events for an arbitrary amount of time, violating the Light-Off-Time-Limit constraint. As before, we simply note the dependency and leave implementation details to the developer.

While simulating *Four Lifts Test*, we discover that Schedule-Lifts' Summons-Button-Event transition is self-dependent. If multiple events occur on Up or Down summons buttons less than Light-On-Time-Limit apart, the Summons-Button-Event transition cannot meet the Light-On-Time-Limit time constraint for the succeeding events. We fixed this dependency by setting Summons-Button-Event delay to half its constraint. This solves the problem for this transition because Summons-Button-Event can turn on more than one summons light simultaneously, handling one or more events on the summons buttons during each execution.

While simulating the *Off State Test*, we discover a dependency between Schedule-Lifts' Status-Event and Lifts-Available transitions. Lift status changes from the Off to On state cause the Status-Event transition to fire, but since the status event occurs while Schedule-Lifts is in the Off state, Status-Event cannot meet the Light-Off-Time-Limit constraint. This problem can be resolved by setting Status-Event-Delay to Light-Off-Time-

Limit minus Lifts-Available-Delay or 99 ms. However, consecutive events on Status less than Light-Off-Time-Limit apart create a dependency between the Status-Event transition and itself. To fix this dependency we must set Status-Event-Delay to one half Light-Off-Time-Limit or 50 ms. Hence, the smallest time constraint for Status-Event-Delay is 50 ms, and we set Status-Event-Delay to 50 ms. This resolves both dependencies for this transition.

During the *All-Summons All Lift Test*, we discover that the lift average response time constraint is quite sensitive to the particulars of the test case. Initially the lifts are all on the first floor in the idle state. If the all-summons test is executed without distributing the lifts throughout the floors in a tall building, the response time is of course very poor. What's more, the time constraint is considerably more dependent on the physical laws governing lift operations (i.e., motor speed, door cycle time, number of floors and lifts, and in real life, how long people hold the lift door open) than it is on the millisecond delays imposed by our state machines. In our simulation, using times from an actual lift in our building, we believe 30 seconds is a more realistic average response constraint than 20 seconds. To fully investigate the average response time constraint requires a more detailed simulation, including implementation of person objects and use of probability distribution functions. While such a simulation is within the capabilities of VHDL, it is beyond the scope of this thesis effort. Instead, having made our point, we change the average response time constraint from 20 to 30 seconds.

6.6 Conclusion

In this chapter we describe applying VHDL to the cruise control and lift controller problems. We show how to drive the VHDL simulation with a testbench, increase simulation power by combining multiple FSMs in a single simulation, and adding other entities. We discuss running the VHDL simulator interactively, via the debugger and in the batch mode.

We describe VHDL test cases, and we demonstrate our success with example specification improvements made as a result of applying VHDL. We note some behavioral improvements and many temporal improvements to both the cruise control and lift controller

specifications. VHDL improvements are largely due to the ability to simulate multiple FSMs, generate more complete test cases, and model asynchronous events.

We note several dependencies that can be resolved by different implementations. We can express these implementations by combining transitions and states or by creating multiple FSMs from a single FSM, but we cannot express orthogonal states in the current R-Spec model. We could expand the V-Spec to accommodate them by specifying a separate concurrent process for each orthogonal substate.

VII. Comparing Reacto and VHDL Results

In Sections 4.4 and 6.5 we discuss the specification improvements resulting from using Reacto and VHDL. Although some results are similar, many are different. These similarities and differences are related to the similarities and differences in Reacto and VHDL capabilities. We depict some relative capability similarities and differences via the Venn diagram in Figure 68.

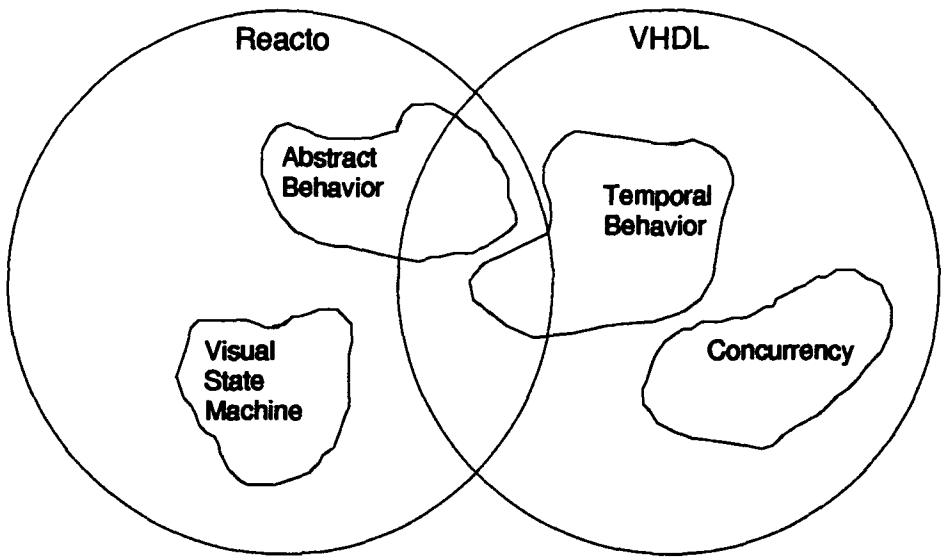

Figure 68. Reacto and VHDL Capabilities

As we apply our methodology to the example problems we exploit these differences and similarities to improve the example SRSs. First, we use Reacto's high level abstractions and state machine visualization to efficiently produce the R-Spec. Second, we use Reacto's verifier and simulation capability to examine and correct many behavioral and some temporal problems. Third, we make use of Reacto and VHDL similarities, providing a transformation from the R-Spec to the V-Spec. Finally, we use VHDL's concurrency and temporal capabilities to fine tune behavioral problems, examine temporal behavior in detail, and expose dependencies between transitions. The differences between the two languages produce some different results, and we now compare some example results. After that, we summarize the benefits and limitations of the two languages.

7.1 *Comparing Reacto and VHDL Activation Allowed Tests*

Figure 69 contains portions[1] of the output from the Reacto cruise control *Activation Allowed Test* and Figure 70 contains portions of VHDL output from the same test case.

```
**Start Activation Allowed Test ******
6000 ms in-top-gear          = T
6000 ms current-speed        = 30.0
6000 ms engine-running       = T
6000 ms activate             = T
6250 ms Voltage              = 0.2.
6250 ms activate             = NIL
7250 ms Voltage              = 1.0.
8250 ms Voltage              = 1.8.
9250 ms Voltage              = 2.6.
10250 ms Voltage              = 3.3999999.
11250 ms Voltage              = 4.0.
11250 ms current-speed        = 29.0
12250 ms Voltage              = 4.8.
13250 ms Voltage              = 5.6000004.
14250 ms Voltage              = 6.0.
Shut-Off Complete
14750 ms Voltage              = 0.0.
**Stop Activation Allowed Test ******
```

Figure 69. Reacto Cruise Control Activation Allowed Test

When we compare the Reacto and VHDL cruise control simulation output, we see that the Reacto and VHDL state machines behave the same under the same stimulus conditions. Up until the current speed event (11250 ms in Reacto, and 16000 ms in the VHDL simulation) the tests and results are equivalent. Because we change the speed input at different times in the simulations, the remaining events also occur at different times, but the values of the cruise control outputs are equivalent. After 14250 ms in the Reacto simulation and 27000 ms in the VHDL simulation there are some additional differences in the output because the Reacto Shut-Off function and the VHDL Shut_Off procedure (see Volume II) we augment the simulations with are different. The VHDL procedure

[1]Lines containing "Go..." are removed.

```
Start Activation Allowed Test ****
   6000 MS In_Top_Gear         : TRUE
   6000 MS Current_Speed       : 30.00
   6000 MS Engine_Running      : TRUE
   6000 MS Activate            : TRUE
   6250 MS Actuator_Voltage    : Event 0.20
   7000 MS Activate            : FALSE
   7250 MS Actuator_Voltage    : Event 1.00
   8250 MS Actuator_Voltage    : Event 1.80
   9250 MS Actuator_Voltage    : Event 2.60
  10250 MS Actuator_Voltage    : Event 3.40
  11250 MS Actuator_Voltage    : Event 4.00
  16000 MS Current_Speed       : 29.00
  17000 MS Actuator_Voltage    : Event 4.80
  18000 MS Actuator_Voltage    : Event 5.60
  19000 MS Actuator_Voltage    : Event 6.00
  27000 MS In_Top_Gear         : FALSE
  27000 MS Braking             : TRUE
  27000 MS Deactivate          : TRUE
  27000 MS Stop_Acceleration   : TRUE
  27000 MS Current_Speed       : 0.00
  27000 MS Engine_Running      : FALSE
  27500 MS Actuator_Voltage    : Event 0.00
  28000 MS Braking             : FALSE
  28000 MS Deactivate          : FALSE
  28000 MS Stop_Acceleration   : FALSE
Stop Activation Allowed Test ****
```

Figure 70. VHDL Cruise Control Activation Allowed Test

operates independently of the state machine, and generates a sequence of events which take the cruise control to the off state and prepare it for the next input sequence. The Reacto Shut-Off function operates in-step with the state machine and cannot generate a sequence of events, therefore the tester must generate the remaining events in the shut off/preparation sequence.

There is another consistent difference between the simulations in general, and that involves asynchronous events. Both Reacto and VHDL test cases include the asynchronous event tests, but the events are actually synchronized with transitions in the Reacto model because we do not implement asynchronous event handling in Reacto.

The test cases for the Reacto and VHDL Lift state machine are similar, but because we create the motor entity in VHDL to simulate the floor-sensor input for us based on the motor-control signal, the test output from the two is different. It is much simpler to let the motor process update the floor-sensor input than to manipulate the Reacto input stream. We do not change the Reacto input file correspondingly for the sake of identical simulation output.

Since we cannot connect Schedule-Lifts and Lifts together in Reacto, we test Schedule-Lifts separately. In VHDL, we do not test Schedule-Lifts separately. Instead, we connect it to four Lifts, saving us the time and trouble of generating manual Lift output to Schedule-Lifts input data.

These changes save considerable time, but they make it difficult to do a line-for-line comparison between the Reacto and VHDL simulations.

7.2 Reacto Benefits and Limitations

Reacto benefits include:

- High Level Abstractions
- Incremental Compilation
- Graphical State Machine Language
- Reacto Verifier

- Automated Transformation Capability

Reacto provides high level abstractions leading to easy and intuitive hierarchical state machine specifications. For example, while creating the Lift R-Spec, we discover that the Lift needs to switch its direction on arrival at the top and bottom floors and when no destinations exist beyond the current location in the current direction and there are destinations in the other direction. It is a simple matter to update the Lift's Calc-Direction function to accommodate the change because of Refine's set quantification and set builder notation. Then we quickly recompile only the Calc-Direction function and reexamine the behavior.

The Lift problem in particular is full of special cases which must be addressed. For example, what if an idle Lift is summoned to a floor, but no one pushes a destination button in the summons direction? Should the Lift simply wait until a button in the correct direction is pressed? We specify that the Lift waits for Timeout time, and then recalculates direction. Because of Reacto's simulation capability and powerful abstractions, we discover and handle this and many other special cases quickly and easily.

Reacto graphical state machine language makes it easier for users to understand the R-Spec. Automating the visual simulation helps during manual simulations, but is slow when simulating from a script file.

Reacto's verifier provides some assurance that the specification says what we want it to say. Operating the verifier is not trivial, and we estimate that another thesis could be written investigating its use to verify our R-Specs.

Combining Reacto's state machine language with our time-modeling augmentations allows us to organize and efficiently maintain real-time SRS information. Because Reacto is implemented in Refine, Refine uses the Reacto grammar to verify the syntax and perform simple semantic checks of our R-Spec SRS, allowing us to confidently transform the R-Spec to VHDL. If we implement the automated transformation function, we can generate VHDL code directly from Refine's knowledge base without having to generate our own lexical analyzer or compiler for some other formal specification language.

Reacto limitations include:

- Unfamiliar Syntax
- Asynchronous Event Restriction
- No Concurrency
- Knowledge Base Conflicts

Although users may understand the visual FSM, a background in set theory, and boolean logic, and exposure to Refine syntax is necessary to understand the Reacto code.

Although Section 4.1.3 discusses options for removing the asynchronous event restriction in Reacto, our current implementations of get-input and the system clock allow us to model only synchronous events.

The current version of Reacto does not allow us to model Concurrent States. This restricts us to specifying pieces of large systems as separate R-Specs (for example the lift controller as FSMs Lift and Schedule-Lifts), and introduces problems because we cannot simulate these separate R-Specs together until after the transformation to VHDL. Additionally, even in smaller FSMs like Lift and Schedule-Lifts, we are not able to resolve all transition dependencies, simply pointing out those which require or some other implementation.

Currently, we are not able to work on both the Lift and Schedule-Lifts V-Specs during the same Reacto session. Both FSMs have On and Off states, and they cannot both be loaded in the knowledge base simultaneously, let alone simulated together. This forces us to kill and restart Reacto to work on one and then the other.

7.3 VHDL Benefits and Limitations

VHDL benefits include:

- Concurrency
- Event driven Simulation
- No Asynchronous Event Restriction
- Alternate Behavioral Specification

Concurrency is perhaps VHDL's biggest benefit. Concurrency enables us to simulate multiple V-Specs and other entities simultaneously, greatly increasing simulation power. Second comes Event driven simulation. VHDL's rich set of operators for expressing time (like signal'last-event) are products of the event driven simulation. Because of concurrency and the event driven simulations, we are able to define the PEM and inject and study asynchronous events.

Another VHDL benefit is the ability to use something besides a FSM to specify behavior. We specify behavior without a FSM for our motorized lift rather than provide manual input saving a great deal of time and effort when simulating multiple lifts simultaneously.

VHDL limitations include:

- Unfamiliar Syntax
- Lack of High Level Abstractions
- Missing Operators
- Strong Typing

Like Reacto, VHDL syntax is not easily understood by inexperienced users. Some exposure and training is necessary to understand it.

Considerable time and effort is required to adapt higher level Reacto abstractions like sets and quantification to VHDL. Similarly, creating operators like Implies and Min complicates the process. Although we could study the Reacto, Refine, and Lisp operations in more detail to guarantee that our implementations function exactly as they do in Reacto, we cannot guarantee that our current implementations do (for example, see Implies in Section 5.4).

VHDL's strong typing causes us problems when transforming from Reacto to VHDL as discussed in Section 5.8.

7.4 Conclusion

In this chapter we discuss the differences in Reacto and VHDL results because of the differences in the languages and simulators. Table 5 summarizes Reacto benefits and limitations, and Table 6 summarizes VHDL benefits and limitations.

Table 5. Reacto Benefits and Limitations

Benefits	Limitations
High Level Abstractions	Unfamiliar Syntax
Incremental Compilation	Asynchronous Event Restriction
Graphical State Machine Language	No Concurrency
Reacto Verifier	Knowledge Base Conflicts
Automated Transformation Capability	

Table 6. VHDL Benefits and Limitations

Benefits	Limitations
Concurrency	Unfamiliar Syntax
Event driven Simulation	Lack of High Level Abstractions
No Asynchronous Event Restriction	Missing Operators
Alternate Behavioral Specification	Strong Typing

Together, the two languages fulfill the following requirements for requirements specification languages we identified in Chapter II:

Abstract the real world well. The Reacto abstractions like sets and quantification allow us to model the lift problem without specifying implementation details like set size limits or set algorithms. Our Reacto augmentations and the VHDL language allow us to model time in an abstract way.

Clearly understood by specifier, implementer, and user. Although the Reacto and VHDL language details require some training and experience to understand, Reacto's graphical state machine provides intuitive understanding of the specification in general.

Support verification that the specification and implementation are equivalent. In addition to execution of the R-Spec and V-Spec test cases, Reacto's verifier promises to be a powerful verification tool.

Easy to modify and manipulate. We assert based on our experience over the last few months that both Reacto and VHDL specifications are relatively easy to modify and manipulate. With the addition of an automatic transformation capability, this can only be easier.

Allow tracing of requirements. Our Reacto augmentations provide a means to organize and manage real-time constraint requirements. Reacto's FSM formalism works like a filing cabinet, allowing us to logically organize virtually all requirements.

Executable specifications. Obviously, both R-Specs and V-Specs are executable.

Support specification of concurrency. Reacto allows specification of concurrency using multiple FSMs; however, it currently does not allow concurrent simulation of FSMs. VHDL on the other hand, allows concurrent simulation of multiple FSMs and other entities as well. Although Reacto currently does not allow specification of concurrent states, we can extend the V-Spec to model them.

Support specification of timing constraints. Using our Reacto augmentations, we can specify stimulus-response and response-response timing constraints in Reacto and VHDL. Our Reacto methodology supports investigating them at a high level, but after transformation to VHDL, we can examine time dependent behavior and timing constraints in detail.

VIII. Conclusions and Recommendations

8.1 Summary

Our problem statement says that we will investigate the feasibility, benefits, and problems associated with formalizing, validating and verifying real-time SRSs using Reacto and the Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL). Additionally, this requires defining a mapping and translation from Reacto to VHDL.

Our original specific objectives were:

1. Define a means to map time constraints into Reacto.
2. Define a means to validate time constraints in Reacto.
3. Specify a transformation from Reacto to VHDL.
4. Specify, verify, simulate, and validate the behavior of a both a cruise control and lift system in Reacto and VHDL.
5. Compare Reacto and VHDL capabilities.
6. Recommend enhancements to Reacto and VHDL.

We have:

1. Added time and timing constraints to Reacto specifications.
2. Formalized cruise control and lift controller SRSs.
3. Validated and verified behavior and timing constraints of the cruise control and the lift controller in Reacto via executable simulations.
4. Defined a behavior preserving mapping between Reacto and VHDL FSMs, especially for the cruise-control, lift, and schedule-lifts R-Specs, and manually transformed the three R-Specs to V-Specs.
5. Validated and verified behavior and timing constraints of three state machine specifications in VHDL making significant improvements in the specifications.
6. Defined a Preemptive Execution Model for VHDL FSMs.
7. Discovered the effects of asynchronous events on the cruise-control, lift, and schedule-lifts behavior in VHDL.

8. Discovered dependencies in concurrently operating Lift and Schedule-Lifts VHDL state machines.
9. Demonstrated the flexibility of the Lift problem decomposition and our methodology by expanding the Lift and Schedule-Lifts simulation to 9 Lifts and 20 floors.

We have met our objectives. We don't claim to have found all possible dependencies in the Cruise and Lift problems, but only in two respects did we hope to do more. One, we hoped to effectively use the Reacto Verifier to prove our R-Spec consistency. The verifier is complicated, and we do not understand its use well enough to write the necessary lemma files to support the verifier. Two, at one point we hoped to provide an automated transformation from Reacto to VHDL. Because of the differences in the Reacto and VHDL languages, defining the target VHDL FSM that models the R-Spec FSM behavior correctly took a great deal of time, but now that it is almost completely defined, creating an automated transformation should be easier.

We have gained a means to formally specify real-time systems, including time constraints as state machines, and verify and validate their behavior. Our methodology may point out dependencies that may not be evident or even a problem with some other specification method, but it does give a framework to expose and evaluate dependencies leading to a clearer specification and better solution than we could have reached without a method at all. Expanding the methodology to allow concurrent states could enable a developer to try out different implementations, leading to a successful solution.

Specification of concurrency cannot and should not be avoided in all cases. We believe it is beneficial to start out with a sequential specification. This allows us to discover the dependencies and annotate them without introducing unnecessary concurrency. This provides the most flexibility to the developer. We assert that it is beneficial to break the lift problem down into two concurrently operating FSMs— e.g., more than one Lift. Within those FSMs, more concurrent behavior could be specified, but we avoid it to stay out of the realm of implementation. Deciding when to specify concurrency must depend on the problem and the objectives.

Concurrency allows us to break our problem down into a collection of state machines and simulate their behavior simultaneously. Concurrency within a single state machine

could be provided by orthogonal states, or by creating two separate state machines operating independently. In the case of the Schedule-Lifts example, the two transitions Summons-Button-Event and Status-Event are dependent, and no sequential solution except combining them into a single transition could eliminate the dependency; that solution is an implementation resolving the natural dependency, no longer a specification. The capability to specify concurrent states would allow us to investigate the problems associated with that solution before building the concurrent implementation. It would also allow the specification to archive the decision to implement concurrency. We have not provided a simple solution to determine when to specify concurrency, and when not to. However, since there are times when concurrency should be specified, the model should accommodate it.

8.2 Recommendations

8.2.1 Reacto Enhancements Allen and Ladkin have defined relations on time intervals (2) (24). Adding the ability to represent time intervals and reason about them in Reacto may be more powerful than the method we have used to specify, verify and validate temporal behavior. Although we intended to use such a formalism to specify behavior as a function of time, it proved to be more than we could accomplish in this masters thesis. We assert that it is worth investigating further, perhaps even augmenting Reacto's verifier to reason about time intervals.

We don't claim to have found all possible dependencies in the Cruise and Lift problems, and we note many detailed test cases could be necessary to fully examine large systems. We hope that expanding Reacto's verification engine to examine transition dependencies and time constraints could eliminate or reduce this burden.

A graphical means to specify the R-Spec FSM's interface to the outside world could be a plus. Something along the lines of an SADT diagram would be beneficial. Perhaps an Intervista windows application could be developed to generate the auxiliary file inputs and outputs for the R-Spec.

Using VHDL attributes like s'last-event, s'last-value, and s'transaction enable us to detect constraint violations in VHDL. Creating attributes for interface variables to support these functions could enhance Reacto's ability to simulate and verify timing constraints.

Similarly, if time becomes an inherent part of Reacto, adding a delay attribute to transitions is important. Also, time constraints could be modeled as attributes of the state object, similar to its *own-vars* attribute. Unless time is implemented as an inherent part of Reacto, it will be difficult to manage. For example, if concurrent states are allowed, how will we solve the problem of updating a global clock when two transitions are executing simultaneously— which value will the clock take on?

We don't currently provide a graphical representation of the timing constraints, although one could label each transition with a delay and show any timers associated with states.

Implementing the automated transformation is a significant task, but even a limited transformation could improve the methodology.

8.2.2 VHDL Enhancements Provide a more general set theoretical data type and set theoretical operations. VHDL text input and output is very cumbersome, it needs better input/output facilities.

Currently, our VHDL FSM is no more concurrent than the Reacto FSM is. If the Reacto model eventually supports orthogonal states, the VHDL machine must be changed to a set of concurrently running processes, one for each orthogonal sub state. This involves using the local signals to communicate between the state machines, and arbitration functions any time two or more states output the same information. The behavior of the resolution function must mimic the resolution defined in Reacto for their state machines when concurrency is implemented. Implementing a monitor in VHDL could help accomplish this.

8.3 Lessons Learned

8.3.1 Reacto Reacto is a very promising tool; its high level abstractions and theorem prover make it unique and powerful. But, Reacto is a new language/tool, not yet fully debugged or user friendly. Since no one here worked with Reacto previously, we learned things the hard way— by trial and error. We hope that those who come after us benefit from our experience, and we note that fixes to many of the following problems are already being implemented at Kestrel Institute.

Changing information that appears in both the graphics windows and in the spec file, like the name of a transition, are best made from the graphics window. First, make a backup spec file copy, then make the changes graphically. Next, save the graphics and the spec from the graphics menu, and then manually update the backup spec file with the changes reflected in the updated one. Finally, copy your manually edited version back over the saved spec. This preserves your comments, capitalization and indentation.

Making changes to the state and transition attributes which are reflected in the spec files requires the input of lisp expressions to the graphical editor. It is far easier to change them in the source file via Emacs, and reload it.

The error message: "error: attempt to call re:*undefined*" which sometimes occurs during simulations is a result of inconsistencies between the graphics and the spec files. The best solution is to copy spec/aux files to a safe place, delete the spec, reinput the graphics, copy the saved copy of the spec/aux files back, and update them with the new transition numbers.

In order to create a new spec by copying an existing spec and adding a prefix to the names, make sure file names include the full name of the highest level state. For example, if cruise-control is your state, the filenames should be cruise-control-spec.re and cruise-control-auxiliary.re or Reacto's copy existing spec option fails.

When you save the spec files from the graphics menu, it saves a pretty-printed copy of the spec from the knowledge base. Since there are no comments in the knowledge base, there are no con ıts in the new spec file.

Occasionally changes made to the source files are not reflected in the knowledge base. If for example, you change the name of a function and recompile, the old function remains in the knowledge base until Refine is killed and restarted. Hence, your system may behave differently, or not work at all after Refine is killed and restarted. Take care, especially when changing or deleting names.

Run under Xwindows NOT Openwindows. Many problems stem from the Intervista interface which Reacto uses for its graphical interface. Examples include extra windows popping up, and lost input in graphics and graphics-editor modes.

Use Emacs as opposed to another editor, it saves you time in the long run, and keeps you from overwriting files with two editors running. Keep backup files! Use Emacs *replace-regular-expression* to search and replace dashes in R-Spec files to underscores in V-Spec files.

It is difficult to create aesthetically pleasing state charts automatically in Reacto. The initial positioning of the transition arrows and labels is not done manually, and you may want to rearrange them. To redraw a transition manually requires focusing on the transition first to avoid mouse-handler errors.

We discovered a couple of problems with version incompatibility between Reacto and Intervista upgrades, but Kestrel Institute's Li-Mei Gilham resolved them quickly and courteously by e-mail. She is very responsive and professional. Her e-mail address is "wu@kestrel.edu".

8.3.2 VHDL VHDL has proven its utility to hardware engineers who use it daily to specify and simulate their systems without going to the expense of building them. We believe it has great potential to do the same for software engineers, especially for real-time systems. Unlike with Reacto, we have the benefit of previous experience with VHDL. VHDL has been around for years, and consequently it is more mature than Reacto. Nevertheless, we pass on some lessons learned.

When VHDL signals are included in a process sensitivity list, inertial signal assignments of the current signal value to those signals is treated like a transport assignment. This results in extra transitions during VHDL simulations because high priority

Current_State signal assignments cannot preempt low priority Current_State signal assignments when the next state for both transitions is the same. We added an extra enumeral *Dummy* to the Current state declaration:

```
type State_Type is (Off, Cruise, Accelerating, Idle, Dummy);
```

and changed the last line of the Cleanup procedure to:

```
Current_State          <= Dummy after D;
Current_State          <= Next_State after D;
```

effectively clearing the Current_State signal driver queue.

VHDL allows users to specify each piece of a system in separate files. For example, one file may contain the entity declaration, another the architecture, and a third file, the configuration. We find it much easier to keep track of changes to these pieces by keeping them in a single file. This also results in far fewer files, and fewer compilation dependency problems.

VHDL allows us to configure the entire Test System (TS) in a single configuration, but for large systems, this can cause the analyzer to abort because of some size limit. It uses less space to configure each piece separately, and combine these smaller configurations together to create the test system. For example, creating the Motorized_Lift configuration, and using it instead of configuring several lifts and motors together in a TS takes much less space, and is simpler to read.

Appendix A. *Cruise Control Test Cases*

A.1 Initialization Test

Initially, the cruise control shall be in an off state. Therefore, upon startup, throttle actuator voltage shall be zero until speed is greater than 30 miles per hour and the driver command activate is asserted. Test steps:

1. Set In-Top-Gear, Engine-Running to true
2. Set Braking to false
3. Set Speed to 0 mph
4. Is Actuator-Voltage 0.0 volts?

A.2 Activation Denied Test

There are three phases to this test. The first phase tests that the system shall not activate when the car is in top gear, the engine is running, but the speed is not 30 mph or more. The second phase tests that the system shall not activate when the car's speed is 30 mph or more, the transmission is in top gear, but the engine isn't running. The final phase tests that the system shall not activate when the cars speed is 30 mph or more, the engine is running, but the transmission isn't in top gear. Requirements Tested: R1, R2, R3. Test steps:

1. Set In-Top-Gear, Engine-Running to true
2. Set Braking to false
3. Set Speed to 29 mph
4. Toggle Activate between true and false
5. Is Actuator-Voltage 0.0 volts?
6. Set Speed to 30 mph
7. Set Engine-Running to false
8. Toggle Activate between true and false
9. Is Actuator-Voltage 0.0 volts?
10. Set Engine-Running to true
11. Set In-Top-Gear to false
12. Toggle Activate between true and false
13. Is Actuator-Voltage 0.0 volts?

A.3 Activation Allowed Test

This tests that the cruise control shall activate when the engine is running, the transmission is in top gear, and the speed is at least 30 mph. Requirements Tested: R1, R2, R3, C4. Test steps:

1. Set In-Top-Gear, Engine-Running to true
2. Set Braking to false
3. Set Speed to 30 mph
4. Toggle Activate between true and false
5. Set Speed to 29 mph
6. Is Actuator-Voltage greater than 0.0 volts?

A.4 Deactivation Test

This tests that the cruise control shall shut off when the driver presses deactivate. Requirements tested: R1, R4. Test steps:

1. Set In-Top-Gear, Engine-Running to true
2. Set Braking to false
3. Set Speed to 30 mph
4. Toggle Activate between true and false
5. Set Start-Accelerating to true
6. Set Deactivate to true
7. Is Actuator-Voltage = 0.0 volts?

A.5 Acceleration Test

This tests that the cruise control shall accelerate the car when the driver presses Start-Accelerating, and that it stops accelerating when the driver presses Stop-Accelerating. It also tests that the acceleration shall be approximately $1 mph/sec$. Requirements Tested: R5, C1, C4, C5, C6, C7, C8. Test steps:

1. Set In-Top-Gear, Engine-Running to true
2. Set Braking to false
3. Set Speed to 30 mph
4. Set Acceleration to 1.0
5. Toggle Activate between true and false
6. Toggle Start-Accelerating between true and false
7. Is Actuator-Voltage greater than 0.0 volts?

8. Increase Acceleration by 1 mph/sec
9. Is Actuator-Voltage constant?
10. Increase Acceleration by 1.2 mph/sec
11. Is Actuator-Voltage 0.0?
12. Increase Acceleration by 0.8 mph/sec
13. Is Actuator-Voltage 8.0?
14. Toggle Stop-Accelerating between true and false
15. Leave Acceleration constant
16. Is Actuator-Voltage greater than 0.0 volts and constant?

A.6 Resume Test

This tests that the cruise control shall resume the previous speed when the driver presses Resume after Braking or not In-Top-Gear. Requirements Tested: R6, R8, R10, R11, R12, C4. Test steps:

1. Set In-Top-Gear, Engine-Running to true
2. Set Braking to false
3. Set Speed to 30 mph
4. Toggle Activate between true and false
5. Is Actuator-Voltage greater than 0.0 volts?
6. Set Braking to true
7. Is Actuator-Voltage = 0.0 volts?
8. Set Braking to false
9. Toggle Resume between true and false
10. Is Actuator-Voltage = previous voltage?
11. Set In-Top-Gear to false
12. Is Actuator-Voltage = 0.0 volts?
13. Set In-Top-Gear to true
14. Toggle Resume between true and false
15. Is Actuator-Voltage = previous voltage?
16. Toggle Deactivate between true and false
17. Toggle Resume between true and false
18. Is Actuator-Voltage = 0.0 volts?

A.7 Downhill Test

This tests that the cruise control shall attempt to maintain the selected speed by decreasing actuator voltage to minimum when the current speed remains more than 2 mph above the selected speed. Requirements Tested: C2, C4. Test steps:

1. Set In-Top-Gear, Engine-Running to true
2. Set Braking to false
3. Set Speed to 45 mph
4. Toggle Activate between true and false
5. Is Actuator-Voltage greater than 0.0 volts?
6. Gradually increase speed to 50 mph
7. does Actuator-Voltage gradually decrease to 0.0 volts?

A.8 Uphill Test

This tests that the cruise control shall attempt to maintain the selected speed by increasing actuator voltage to maximum when the current speed remains more than 2 mph below the selected speed. Requirements Tested: C3, C4. Test steps:

1. Set In-Top-Gear, Engine-Running to true
2. Set Braking to false
3. Set Speed to 45 mph
4. Toggle Activate between true and false
5. Is Actuator-Voltage greater than 0.0 volts?
6. Gradually decrease speed to 40 mph
7. does Actuator-Voltage gradually increase to 8.0 volts?

A.9 Breaking During Activate Delay Test

This test is an asynchronous event test, meant for use with VHDL, and explicitly designed to expose timing dependencies between the Startup and Not-Enabled transitions. It models the situation where the driver activates the cruise control, and less than 250 ms later puts on the brakes. We expect the cruise control to output 0 volts within 500 ms of the braking event. Test steps:

1. Set In-Top-Gear, Engine-Running to true
2. Set Braking to false
3. Set Speed to 45 mph
4. Set Activate true
5. Set Braking true 100 ms after Activate
6. Is Actuator-Voltage 0.0 volts within 500 ms of braking?

A.10 Breaking During Activate Asserted Test

This test is an asynchronous event test, meant for use with VHDL, and explicitly designed to expose timing dependencies between the Startup and Not-Enabled transitions. It is similar to the Breaking During Activate Delay Test, except for the fact that the braking event occurs more than 250 ms after the activate event, while activate is still asserted. We expect the cruise control to turn off within 500 ms of the braking event. Test steps:

1. Set In-Top-Gear, Engine-Running to true
2. Set Braking to false
3. Set Speed to 45 mph
4. Set Activate true
5. Set Braking true 450 ms after Activate
6. Is Actuator-Voltage 0.0 volts within 500 ms of braking?

A.11 Breaking After Activate B4 Cruise Test

This test is an asynchronous event test, meant for use with VHDL, and explicitly designed to expose timing dependencies between the Startup and Not-Enabled transitions. Similar to the two previous test cases, this test case examines the behavior of the cruise control when the driver presses and releases the activate button very quickly and puts on the brakes within 250 ms of pushing the activate button. Again, we expect the cruise control to output zero volts within 500 ms of the braking event. Test steps:

1. Set In-Top-Gear, Engine-Running to true
2. Set Braking to false
3. Set Speed to 45 mph
4. Set Activate true, then false after 150 ms
5. Set Braking true 175 ms after Activate is true
6. Is Actuator-Voltage 0.0 volts within 500 ms of braking?

A.12 Resume During Breaking Test

This test is an asynchronous event test, meant for use with VHDL, and explicitly designed to expose timing dependencies between the Resume-And-Enabled and Not-Enabled transitions. It models the situation where the driver activates the cruise control, puts on the brakes after the cruise control activates, and while the brakes are pressed, pushes the resume button, releasing the resume button and the brakes simultaneously. We expect the cruise control to continue outputting zero volts within 500 ms of the braking event even though the resume button is pressed. Test steps:

1. Set In-Top-Gear, Engine-Running to true

2. Set Braking to false
3. Set Speed to 45 mph
4. Set Activate true, then false after 150 ms
5. Set Braking true 1750 ms after Activate is true
6. Set Resume true 2505 ms after Activate is true
7. Is Actuator-Voltage 0.0 volts within 500 ms of braking?

A.13 Deactivate Overlaps Resume Test

This test is an asynchronous event test, meant for use with VHDL, and explicitly designed to expose timing dependencies between the Shutdown , Not-Enabled, and Resume-And-Enabled transitions. It models the situation where the driver activates the cruise control, subsequently deactivates it, and while the deactivate button is pressed and before the cruise control actually shuts down, also pushes the resume button, keeping it pressed until after the deactivate button is released. We expect the cruise control to continue outputting zero volts within 500 ms of the deactivate event even though the resume button is pressed. Test steps:

1. Set In-Top-Gear, Engine-Running to true
2. Set Braking to false
3. Set Speed to 45 mph
4. Set Activate true, then false after 455 ms
5. Set Deactivate true 1750 ms after Activate is true, false after 2200 ms
6. Set Resume true 1800 ms after Activate is true, false after 3009 ms
7. Is Actuator-Voltage 0.0 volts within 500 ms of Deactivate?

Appendix B. *Lift Controller Test Cases*

B.1 Lift Test Cases

B.1.1 All Summons Test This test tests lift's ability to handle all Summons from every floor. In the real world, this corresponds to people simultaneously pushing every summons button on every floor requesting the elevator pick them up. Lift shall stay on the first floor until the (1 UP) Summons is removed from Summons and Outstanding Requests, then go up, stop at each floor, wait until the up Summons for that floor is removed from Summons and Outstanding Requests, until it reaches the top floor. From the top floor the lift shall go down, stop at each floor until it reaches the second floor, where it goes Idle after (2, DN) is removed from Summons and Outstanding Requests.

B.1.2 All Destinations Test This test tests lift's ability to handle people inside the lift simultaneously pushing all destination buttons. The lift shall initially turn on all destination lights, and since it is already at the second floor, the lift shall subsequently turn off the second floor light. Since it was going down when it stopped last, it shall continue to go down, stop at the first floor, and extinguish destination light one. Before it gets to the first floor, a passenger pushes the second floor destination button, and now the lift shall turn on destination light number 2. At the first floor, it shall reverse direction, go up, stop at each floor, and extinguish the destination light for each floor as it arrives. After leaving the second floor another passenger pushes the second floor button, and the lift shall turn on floor 2's destination light. The lift shall continue to the top floor, and return to the second floor without stopping at the third floor, extinguish destination light 2, and go Idle.

B.1.3 Emergency Button Test This test tests lift's response to emergency button events. Initially, we push the emergency button while the lift is Idle, to see if it goes to the Off state. Then we release the emergency button, and the lift shall return to the Idle state. Next we push the emergency button again, and after the lift goes to the Off state, we push destination button 4. Lift shall turn on destination light 4, since we specify that it schedules destinations even while stopped for an emergency. When we release the emergency button, the lift shall start moving toward the fourth floor. While the lift is between floors, we push the emergency button again, and lift shall not stop until it reaches a floor, and it shall stop upon reaching any floor whether it is in the schedule or not. When we release the emergency button after the lift stops, it shall resume moving, stop when it gets to the 4th floor, extinguish the 4th floor destination light, and go Idle.

B.1.4 Mixed Destinations and Summons Test This test tests lifts ability to handle both summons requests and destination requests together. First, we summon lift to take us down from the second floor. Since it is at the fourth floor, lift shall start moving down, and stop at the second floor and go Idle after (2 DOWN) is removed from Summons and Outstanding-Requests. After the lift arrives at the second floor, we press destination button 1, and the lift shall turn on destination light 1, and move down to the first floor. Before the lift arrives at the first floor, we input a summons request (3 DOWN). The lift

shall continue moving to the first floor, stop, reverse direction and head for the third floor. After the lift leaves the first floor, and before it gets to the second floor, we input another summons request for (2 UP). The lift shall stop at the second floor, wait until (2 UP) is removed from Summons and Outstanding-Requests, move to the third floor, and wait until (3 DOWN) is removed from Summons and Outstanding-Requests. Now, we press destination button 1, and the lift shall turn on destination light 1, take us to the first floor, stop and go Idle.

B.1.5 Timeout Test Timeout tests the lift against the response-response constraint which specifies the amount of time the lift waits for a destination in the current direction if there are no destinations already in that direction. It tasks the lift to satisfy both up and down summons from third floor. But no one gets on to push destination requests. Lift should satisfy the up summons first, timeout, satisfy the down summons, then go idle.

B.2 Schedule Lifts Test Cases

B.2.1 Off State Test This test tests schedule lifts behavior when all lifts are off. The scheduler should transition to the Off state and ignore destination button requests until at least one lift is restored.

B.2.2 All Summons 1 Lift Test This test tests schedule lifts behavior when only one lift is available to handle summons requests from all floors in all directions. Lift 2 is the only active lift, it should be tasked to handle all summons. As it cycles visiting all floors in all directions, summons should be removed from outstanding requests as floors are visited. Finally, outstanding requests should be nil.

B.2.3 Idle Schedule Test This test tests schedule lifts behavior when all elevators are available, and only one is idle. It should task the one lift with every summons unless there is an elevator at the correct floor going the right way.

B.2.4 All Summons Test This test tests schedule lifts behavior when all lifts are available to handle summons requests from all floors in all directions. Schedule lifts should distribute the summons requests to lifts fairly, depending on all lift status information.

Appendix C. *Reacto Input and Output*

C.1 Input

Reacto *Interface Variables* provide a way for the R-Spec to communicate to the R-Spec from the external environment (23:7). Users define Interface Variables in R-Spec auxiliary files during Developer editing sessions. Interface variables have the following user defined attributes:

- name
- interface-var-base-type
- buffered?
- producer-function

The interface variable *name* attribute is used to refer to the interface variable in the R-Spec (23:7).

The *interface-var-base-type* attribute defines the type of the interface variable (23:7). It may be any Refine predefined data type or user defined type.

The interface variable *buffered?* attribute is a boolean flag. If buffered? is true, the interface variable acts like a queue[1]. Actually, a buffered interface variable is a sequence of type interface-var-base-type. If buffered? is false, the interface variable is not a sequence, only a single element of type interface-var-base-type (23:7).

The *producer-function* attribute specifies the name of a producer function for the interface variable. Producer functions have no parameters, and they must return a value of type interface-var-base-type (23:7).

There are three predefined interface variables, *keyboard-input*, *file-input*, and *keyboard-char-input*. Each predefined interface variable has a predefined producer function. FSMs use *keyboard-input* and *file-input* to read lisp objects from the keyboard

[1] External sources write to the tail of the queue, the R-Spec FSM reads from the head of the queue

and input files via the Refine compiler. The *keyboard-char-input* provides a way to read a single character from the system keyboard (23:8).

Reacto has four system defined functions which R-Spec FSMs use to access interface variables.

- empty-interface-variable?
- clear-interface-variable
- examine-interface-variable
- read-interface-variable

The *empty-interface-variable?* function is a boolean function that returns true when the interface variable is undefined or the null sequence (23:9).

The *clear-interface-variable* function sets unbuffered interface variables to undefined and removes one element from the head of a buffered interface variable queue. Calling clear-interface-variable when the interface variable is undefined or the null sequence generates an error (23:9).

The *examine-interface-variable* function returns the value of unbuffered interface variables or the value of the first element of buffered interface variables. When the interface variable is undefined or the null sequence, Reacto calls the interface variable's producer function and returns the new value. Calling examine-interface-variable on interface variables without a producer function when the interface variable is undefined or the null sequence generates an error (23:9).

Calling *read-interface-variable* function is like calling examine-interface-variable and clear-interface-variable one after the other (23:9).

Clear-interface-variable and read-interface-variable can only be called from transition actions, while empty-interface-variable? and examine-interface-variable may also be used in state assertions and transition predicates (23:9).

C.2 Output

The Reacto predefined function *update-screen* provides users a way to write string output to a 26 line by 40 character window during simulations. Users control output format by passing cursor coordinates to update-screen (23:9).

Bibliography

1. Alagar, V.S. and G. Ramanathan. "Functional Specification and Proof of Correctness for Time Dependent Behavior of Reactive Systems," *Formal Aspects of Computing*, *03*(3):253–283 (Jul-Sep 1991).
2. Allen, James F. "Maintaining Knowledge about Temporal Intervals," *Communications of the ACM*, *26*(11):832–843 (November 1983).
3. Allen, James F. "Towards a General Theory of Action and Time," *Artificial Intelligence*, *23*(2):123–154 (July 1984).
4. Bailor, Maj Paul D. "Program Transformation Systems." Class handout for CSCE 595, Software Generation and Maintenance, Winter Quarter 1992.
5. Balzer, Robert and Neil Goldman. "Principles of Good Software Specification and their Implications for Specification Language," *IEEE Conference on Specifications of Reliable Software*, 7(2):58–67 (March 1979).
6. Barton, David L. "A First Course in VHDL," *Design Automation Guide*, 40–47 (1988).
7. Connor, Michael F. "SADT - Structured Analysis and Design Technique Introduction." *1980 International Engineering Management Conference Record*. 138–143. New York: IEEE Press, 1980.
8. Dasarathy, B. "Timing Constraints of Real-Time Systems: Constructs for Expressing Them, Methods of Validating Them," *IEEE Transactions on Software Engineering*, *11*(1):80–86 (January 1985).
9. Davis, Alan M. *Software Requirements : Analysis and Specification*. Englewood Cliffs NJ: Prentice Hall, 1990.
10. Douglass, Randall L. *Formalization and Validation of an SADT Specification Through Executable Simulation Using the Refine Specification Environment*. MS thesis, AFIT/GCS/ENG/91D-5, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1991.
11. Drusinsky, Doron and David Harel. "Using Statecharts for Hardware Description and Synthesis," *IEEE Transactions on Computer-Aided Design*, *8*(7):798–807 (July 1989).
12. Eickmeier, Daniel L. *Formalization and Validation of an SADT Specification Through Executable Simulation in VHDL*. MS thesis, AFIT/GCS/ENG/91D-6, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1991.
13. Gilham, Li-Mei. "Toward Reliable Reactive Systems." *Appeared in Proceedings of the 5th International Workshop on Software Specification and Design, May 1989*. 1–8. Palo Alto, CA: Kestrel Institute, February 1989.
14. Harel, David. "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming*, *8*:231–274 (1987).

15. Harel, David. "On Visual Formalisms," *Communications of the ACM*, *31*(5):514–530 (May 1988).

16. Harel, David et al. "STATEMATE: A Working Environment for the Development of Complex Reactive Systems," *IEEE Transactions on Software Engineering*, *16*(4):403–414 (April 1990).

17. Hatley, Derek J. and Imatiaz A. Pirbhai. *Strategies For Real-Time System Specification*. New York: Dorset House Publishing, 1987.

18. Ibrahim, Rosalind L., et al. "Should Concurrency be Specified?." *Specification and Verificaiton of Concurrent Systems. Proceedings BCS-FACS workshop (TR-45)*. 246–271. Stirling UK: Stirling, July 1988.

19. IEEE Press, New York. *IEEE Standard VHDL Language Reference Manual - IEEE Std 1076-1987*, 1988.

20. Jahanian, Farnam. "Verifying Properties of Systems with Variable Timing Constraints." *Proceedings, Real Time Systems Symposium(Cat No.89CH2803-5)*. 319–328. New York: IEEE Computer Society Press, December 1989.

21. Jahanian, Farnam and Aloysius Ka-Lau Mok. "Safety Analysis of Timing Properties in Real-Time Systems," *IEEE Transactions on Software Engineering*, *12*(9):890–904 (September 1986).

22. Kestrel Institute. *Reacto Verifier User's Guide*, 1990. Version 1.0.

23. Kestrel Institute. *Reacto Users Manual*, 1992. Version 2.0.

24. Ladkin, Peter. "Time Representation: A Taxonomy of Interval Relations." *Proceedings of AAAI-86*. 360–366. 1986.

25. Levi, Shem-Tov and Ashok K. Agrawala. *Real-Time System Design*. New York: McGraw-Hill Book Company, 1990.

26. Lipsett, Roger et al. *VHDL: Hardware Description and Design*. Boston: Kluwer Academic Publishers, 1989.

27. March, Stephen G. *An Object Oriented Analysis Method for Ada and Embedded Systems*. MS thesis, AFIT/GCS/ENC/89D-1, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1989.

28. Nejad-Sattary, M. and P.E. Osmon. "A notation for Real-Time System Specification." *Proceedings of the UK IT 1990 Conference*. 358–364. London: IEE, March 1990.

29. Pressman, Roger S. *Software Engineering: A Practitioner's Approach*. New York: McGraw-Hill Book Company, 1987.

30. Reasoning Systems Inc. *Refine User's Guide*, 1985. Version 3.0, revised 1990.

31. Ross, Douglas T. "Structured Analysis (SA): A Language for Communicating Ideas," *IEEE Transactions on Software Engineering*, *1*:16–34 (January 1977).

32. Ross, Douglas T. "Applications and Extensions of SADT," *IEEE Computer*, *18*(4):25–35 (April 1985).

33. Ross, Douglas T. and K. Schoman. "Structured Analysis for Requirements Definition," *IEEE Transactions on Software Engineering*, *1*:6–15 (January 1977).

34. Smith, Sharon L. and Susan L. Gerhart. "STATEMATE and Cruise Control: A Case Study." *Proceedings of the Twelfth Annual International Computer Software and Applications Conference (COMPSAC 88)*. 49–56. New York: IEEE, October 1988.

35. Sommerville, Ian. *Software Engineering*. Workingham, England: Addison-Wesley, 1989.

36. Spicer, Kelly L. *Mapping an Object Oriented Requirements Analysis to a Design Architecture that Supports Design and Component Reuse*. MS thesis, AFIT/GCS/ENG/90D-13, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1990.

37. Stankovic, John A. "Misconceptions About Real-Time Computing," *IEEE Computer*, *10*:10–19 (October 1988).

38. Tse, T.H. and L. Pong. "An Examination of Requirements Specification Languages," *The Computer Journal*, *34*:143–152 (April 1991).

39. Wood, William G. "Temporal Logic Case Study." *Lecture Notes in Computer Science, Automatic Verification Methods for Finite State Systems International Workshop, Grenoble France, Proceedings*. 257–263. Berlin: Springer-Verlag, June 1989.

40. Yourdon, Edward. *Modern Structured Analysis*. Englewood Cliffs NJ: Prentice-Hall, 1989.

41. Zave, Pamela. "An Insider's Evaluation of PAISLey," *IEEE Transactions on Software Engineering*, *17*(3):212–225 (March 1991).

42. Zave, Pamela and Daniel Jackson. "Practical Specification Techniques for Control-Oriented Systems." *Proceedings of the IFIP 11th World Computer Congress (Information Processing 89)*. 83–88. New York: North-Holland, September 1989.

43. Zave, Pamela and William Schell. "Salient Features of an Executable Specification Language and Its Environment," *IEEE Transactions on Software Engineering*, *SE-12*(2):312–326 (February 1986).

Vita

Captain Frank Charles Duane Young was born on 22 April 1955 in Choteau, Montana. He graduated from Choteau High School in 1973 and attended Carroll College, Helena, Montana from 1973 to 1974. He enlisted in the United States Air force in October 1974. From January 1975 to October 1979 Capt Young worked as an enlisted computer operator at Air Force Global Weather Central, Offutt AFB, Nebraska. From October 1979 to August 1984, Captain Young was a computer operator and computer systems manager in the base data processing facility at Malmstrom AFB, Montana. From September 1984 to June 1987, Capt Young attended Montana State University via the Air Force's Airman's Education and Commissioning Program. He graduated Magna Cum Laude with a Bachelors Degree in Computer Science. From July 1987 to October 1987, Captain Young attended the Air Force Officer Training School; he was commissioned on 1 October 1987. After a short Technical School assignment at Keesler AFB Mississippi, Captain Young was the Deputy Chief of Computer Operations in Cheyenne Mountain Complex, Colorado Springs, Colorado from March 1988 to August 1990. From September 1990 to May 1991, Captain Young was a Software Engineer and Programmer Analyst on a software development project for a new Cheyenne Mountain system. In May 1991, Capt Young entered the Air Force Institute of Technology in pursuit of a Master of Science degree in Computer Engineering.

Permanent address: 110 1st Ave. S.W.
Choteau, Mt 59422

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1992	Master's Thesis

4. TITLE AND SUBTITLE
Formalizing, Validating, And Verifying
Real-Time System Requirements
With Reacto And VHDL

5. FUNDING NUMBERS

6. AUTHOR(S)
Frank C. D. Young, Captain, USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology, WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GCS/ENG/92D-24

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)
We develop a methodology for formalizing, verifying, and validating the requirements specification of real-time systems based on a graphical and formal hierarchical Finite State Machine (FSM) language *Reacto*. We define a means to quantify time and express real-time constraints in Reacto and a transformation from Reacto to the Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL). Reacto's high level abstractions, graphical nature, and theorem prover produce efficient, accurate, and easily understood specifications. We use VHDL's event driven simulation capability, concurrency, and temporal operators to thoroughly examine temporal dependencies between the state machine transitions, and to increase simulation power by simulating multiple communicating FSMs. We apply the methodology to two example problems, a cruise control, and a lift (elevator) controller. We verify that the state machine specification is consistent and validate the specification using executable simulations in both Reacto and VHDL. We evaluate the methodology against criteria for real-time specification languages and conclude that Reacto and VHDL complement each other well. Together, they abstract the real world well, are clearly understood, verify that the specification and implementation are consistent, are easy to modify, allow requirements tracing, and finally, support specification of concurrency and timing constraints.

14. SUBJECT TERMS
Requirements Analysis, Real Time, System Specification, Reacto, VHDL, Software Engineering, Formal Methods

15. NUMBER OF PAGES
172

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL